ELSEVIER

Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Artificial Intelligence 147 (2003) 119–162

Artificial Intelligence

www.elsevier.com/locate/artint

Contingent planning under uncertainty via stochastic satisfiability

Stephen M. Majercik [a,*], Michael L. Littman [b]

[a] *Bowdoin College, Brunswick, ME 04011-8486, USA*
[b] *Rutgers University, Piscataway, NJ, USA*

Received 22 June 2001

Abstract

We describe a new planning technique that efficiently solves probabilistic propositional contingent planning problems by converting them into instances of stochastic satisfiability (SSAT) and solving these problems instead. We make fundamental contributions in two areas: the solution of SSAT problems and the solution of stochastic planning problems. This is the first work extending the planning-as-satisfiability paradigm to stochastic domains. Our planner, ZANDER, can solve arbitrary, goal-oriented, finite-horizon partially observable Markov decision processes (POMDPs). An empirical study comparing ZANDER to seven other leading planners shows that its performance is competitive on a range of problems.

Keywords: Probabilistic planning; Partially observable Markov decision processes; Decision-theoretic planning; Planning-as-satisfiability; Stochastic satisfiability; Contingent planning; Uncertainty; Incomplete knowledge; Probability of success

1. Introduction

Planning—making a sequence of action choices to achieve a goal—has been an important area of artificial intelligence (AI) research since the field began and this prominence is not difficult to explain. First, the need to plan is pervasive; to a greater or lesser extent, *all* problems can be characterized as planning problems: how should one act (bring resources to bear) to change an existing state into a more desired state? The ability to act in a goal-directed fashion is critical to any notion of intelligent agency.

* Corresponding author.
E-mail addresses: smajerci@bowdoin.edu (S.M. Majercik), mlittman@cs.rutgers.edu (M.L. Littman).

0004-3702/03/$ – see front matter
doi:10.1016/S0004-3702(02)00379-X

Second, planning is an extremely hard problem. Deterministic STRIPS planning (arguably the "easiest" type of propositional planning that is still capable of expressing interesting problems) is PSPACE-complete [13]; unrestricted probabilistic propositional planning in partially observable domains is undecidable [46].

Traditionally, the decision-making models that have been studied in AI planning admit no uncertainty: every aspect of the world that is relevant to the generation and execution of a plan is known to the agent in advance. This unrealistic assumption has been a major impediment to the practical use of AI planning techniques, and there has been a great deal of research in the past decade to create planning techniques that are capable of handling uncertainty in the environment (uncertain initial conditions, probabilistic effects of actions, and uncertain state estimation). One of the attractive features of AI planning is its ability, in some cases, to operate in large domains ($\sim 10^{20}$ states). One reason for this ability is that AI planning typically uses a problem representation that allows significant features of the problem states and actions to be exploited by the solution method.

Researchers in operations research (OR) have studied a planning formalism that directly addresses uncertainty—Markov decision processes (MDPs) and partially observable Markov decision processes (POMDPs). Classical OR algorithms, however, use an impoverished problem representation that does not capture relationships among states, and these techniques are capable of solving problems only in relatively small domains ($\sim 10^6$ states for exact MDP solution methods and many fewer for exact POMDP solution methods in typical domains).

Our work investigates the potential of merging the best characteristics of AI planning (large domains) and OR planning (stochastic domains) to produce a system that can reason efficiently about plans in complex, uncertain applications. The planners we have developed are rooted in the planning-as-satisfiability paradigm. In this paradigm, the planning problem is converted into a satisfiability problem and the efficient solution of the resulting satisfiability problem produces the required plan. This work is inspired in large measure by the success of SATPLAN, a similar planning technique for deterministic domains [34] that encodes the planning problem as a Boolean satisfiability problem and uses stochastic local search to solve the resulting satisfiability problem.

There are significant problems in developing a probabilistic version of SATPLAN. Plans in a stochastic domain can be very complex. Unlike plans in a deterministic setting, optimal plans in a stochastic domain frequently require contingent branches that specify different actions depending on the stochastic outcome of the current action, or loops that repeat an action until a desired result is achieved. In addition, evaluating plans in stochastic domains is difficult. In the deterministic setting, plan evaluation can be accomplished by executing the plan and checking the single execution trace to see whether the final state is a goal state. In the stochastic setting, the uncertainty in the domain means that, in general, there will be multiple possible execution traces for a given plan, with some subset of these traces ending in a goal state. For this reason, plan evaluation requires the equivalent of checking each possible execution trace and summing the probability of each trace whose final state is a goal state.

The main contribution of our research is to show that the planning-as-satisfiability paradigm can be successfully extended to support contingent planning in partially observable stochastic domains. To our knowledge, ours is the only existing planner that

augments the planning-as-satisfiability paradigm to support stochastic domains. ZANDER, the most advanced planner we developed and the one described here, can solve arbitrary, goal-oriented, finite-horizon partially observable Markov decision processes (POMDPs).[1] An empirical study comparing ZANDER's performance to that of seven other leading probabilistic planners—a dynamic programming POMDP algorithm, MAHINUR, SENSORY GRAPHPLAN (SGP), PGRAPHPLAN, SPUDD (stochastic planning using decision diagrams), GPT (general planning tool), and HANSEN-FENG—shows that ZANDER's performance is competitive on a range of problems.

2. Background

This section presents the research context in which we developed our planner.

2.1. Deterministic planning

Informally, a deterministic planning problem is characterized by a finite set of states that the planning agent could find itself in, a finite set of operators, or actions, that transform states to states deterministically, a designated initial state, and a set of goal states. A solution to the planning problem is a sequence of actions that transforms the initial state into one of the goal states.

In recent years, two planning methods based on constraint satisfaction—GRAPHPLAN and SATPLAN—have received a great deal of attention in the planning research community. Both GRAPHPLAN and SATPLAN make use of the notion of search through plan space by considering, in a sense, *all* plans up to a certain length *simultaneously* and attempting to extract a successful plan from this collection.

GRAPHPLAN [5] works by creating a *planning graph* that interleaves layers of nodes representing the status of propositions at a time step with layers of nodes representing possible actions at a time step. Edges in this directed, leveled graph connect actions to their preconditions and their add and delete effects, thus indicating all feasible actions at each time step and their impact on the domain propositions. GRAPHPLAN operates by constructing a planning graph forward from the initial conditions until a layer of propositions appears that contains all the goal propositions. The planner then searches for a plan using backward chaining; if none is found it extends the graph another time step and the search is repeated. The key element of GRAPHPLAN is a scheme for efficiently identifying and propagating pairwise inconsistencies (e.g., two actions that cannot be executed at the same time). GRAPHPLAN outperforms a state-of-the-art planner called UCPOP on several natural and artificial planning problems [5]; it remains one of the best current planners and research on this paradigm is quite active.

SATPLAN [33,34] works by first converting the bounded-horizon planning problem to a propositional satisfiability problem and then using stochastic local search to solve

[1] Since reward-oriented POMDPs can be expressed as probabilistic goal-oriented POMDPs, ZANDER could be applied to arbitrary POMDPs; see note in Section 6.1.

the resulting satisfiability problem. Kautz and Selman [34], in an early paper describing SATPLAN, argue that the planning community, in rejecting general reasoning systems in favor of specialized planning algorithms, learned the wrong lesson from the failure of Green's theorem-proving problem solver. They argue that the lesson to be learned was not that general reasoning systems are inappropriate for planning but that first-order deductive theorem-proving does not scale well. In contrast, propositional satisfiability testing has great potential as a tool for reasoning about plans.

Briefly, SATPLAN converts a deterministic planning problem to a Boolean satisfiability problem by constructing a Boolean formula in conjunctive normal form (CNF) that has the property that any satisfying assignment to the variables in the formula—any *model*—corresponds to a plan that achieves the goal. The satisfiability of the resulting CNF formula is determined using WALKSAT, a generic satisfiability algorithm based on stochastic local search. It is worth noting here that although SATPLAN uses stochastic local search, other satisfiability testing algorithms have been used in the context of planning. The original Davis–Putnam procedure for satisfiability testing [17] uses *resolution* as a key algorithmic component. Resolution was later replaced by *variable splitting* [16], and this latter procedure has completely overshadowed the earlier version. Other systematic solvers that incorporate efficient data structures (SATO, [73]), better heuristics (SATZ, [42]), and constraint satisfaction solution techniques (RELSAT, [3]) have been developed more recently. BLACKBOX [35,37] integrates several of these approaches—WALKSAT (stochastic local search), SATZ, and RELSAT—in a planning system that allows the user to try different solvers on the SAT encoding of a planning problem. Although stochastic local search generally outperforms systematic satisfiability testers by an order of magnitude or more on hard *random* satisfiability problems, there is some evidence that the systematic testers are competitive with stochastic local search on more structured, real-world planning problems [3]. We use a modified version of the Davis–Putnam–Logemann–Loveland satisfiability tester [16] in our planner (Section 5.1).

There are a number of advantages to the planning-as-satisfiability approach. The expressiveness of Boolean satisfiability allows us to construct a very general planning framework. It is relatively straightforward to express planning problems in the framework of propositional satisfiability and this framework makes it easy to add constraints to the planning problem (such as domain-specific knowledge, [36]) to improve the efficiency of the planner. Another advantage echoes the intuition behind reduced instruction set computers; we wish to translate planning problems into satisfiability problems for which we can develop highly optimized solution techniques using a small number of extremely efficient operations. Supporting this goal is the fact that satisfiability is a fundamental problem in computer science and, as such, has been studied intensively. Numerous heuristics and solution techniques have been developed to solve satisfiability problems as efficiently as possible.

There are disadvantages to this approach. Problems that can be compactly expressed in representations used by other planning techniques often suffer a significant blowup in size when encoded as Boolean satisfiability problems, degrading the planner's performance. Automatically producing maximally efficient plan encodings is a difficult unsolved problem. In addition, translating the planning problem into a satisfiability problem may obscure the structure of the problem, making it difficult to use one's knowledge of and

intuition about the planning process to develop search control heuristics or prune plans. This issue has also been addressed; Kautz and Selman [36], for example, report impressive performance gains resulting from the incorporation of domain-specific heuristic axioms in the SAT encodings of deterministic planning problems.

Planning as satisfiability has been an active area of research. Researchers have looked at the issues that arise in connection with efficient conversion of planning problems to satisfiability problems [20,31], improving systematic satisfiability testers [3, 42], understanding and improving stochastic local search [32,54,68], accelerating the search for a plan by including domain-specific knowledge [36], and incorporating the various constraint satisfaction planning techniques in a single planning system [35,37].

2.2. Probabilistic planning

Like a deterministic planning problem, a probabilistic planning problem is specified by a finite set of states, a finite set of actions, an initial state, and a set of goal states. In a probabilistic domain, however, actions transform states to states probabilistically; for a given state and action, there is a probability distribution over possible next states. The solution to a probabilistic planning problem is an action selection mechanism for the planning domain that reaches a goal state with sufficiently high probability. Probability of success is not the only objective that makes sense to consider; other possible objectives include minimizing the length or size of the plan, or maximizing the expected utility achieved by the plan (if there is a *utility function* that assigns a numerical value to each component of the goal, thus providing a quantitative measure of the importance, or *utility*, of each goal component). In our work, we focused on finding plans that maximize the probability of reaching a goal state given a fixed number of plan steps (finite horizon).

The defining characteristic of probabilistic planning is that the actions are probabilistic; the outcome of an action in a given state is a probability distribution over possible next states. There is another type of nondeterministic planning that is relevant in this review, however. It is possible to frame planning problems using *non-probabilistic* actions.[2] A non-probabilistic action can have multiple possible outcomes that depend only on the state in which the action is executed. The effect of the action is deterministic given the state in which it is executed, but the agent may not know *a priori* the state in which it will be executing the action and, hence, its effect. Thus, the uncertainty is represented as a list of possible state/outcome pairs, rather than as a probability distribution over possible outcomes.

A simple example will clarify this distinction between probabilistic actions and non-probabilistic actions. A probabilistic action move(a,b,c) in a blocks-world domain (i.e., move block a off of block b onto block c) might specify that the action is successful with probability 0.85, that block a ends up on the table with probability 0.10, and that nothing happens with probability 0.05. A non-probabilistic version of the same action might specify that if the gripper is functioning and dry, the action will succeed, if the

[2] Such actions have historically been called *conditional* actions [24,25,62]. In our taxonomy of planning under uncertainty, however, we wish to make a distinction between the type of planning and the type of actions used, so we will use the term *non-probabilistic action* to avoid confusion.

gripper is functioning, but wet, block a will end up on the table, and if the gripper is not functioning, nothing will happen. We are concerned here with the former type of action.

The type of planning an agent engages in is, in this sense, a function of the agent's knowledge about the domain. A probability distribution over possible outcomes of an action may, in some cases, be a substitute for better domain knowledge. In the blocks world example, the agent may not know that the move action fails sometimes because the gripper is wet. But experience may allow the agent to estimate a probability distribution over outcomes of that action. Or it may be the case, to extend this example further, that the agent knows that when the gripper is wet, the action *usually* fails, but that with probability 0.05 it succeeds. If the agent does not know *why* the action sometimes succeeds, the agent may still be able to attach a probability distribution to the execution of the action, and plan using that probability distribution.

We will also make a distinction between *conditional planning* and *contingent planning*. In conditional planning, the effects, but not the execution, of actions are contingent on the outcomes of previous actions. In contingent planning, both the effects and execution of actions are contingent on the outcomes of previous actions.[3] Thus, in contingent planning, the agent can make observations and construct a branching plan in which actions are made contingent on these observations. Without the ability to observe its environment and condition its actions accordingly, an agent can only execute a *straight-line plan*, a simple non-contingent sequence of actions, and hope for the best. Such a plan can also be called "open loop", in contrast to "closed loop" plans that condition action choices on run-time observations.

These two distinctions (conditional planning *v.* contingent planning and non-probabilistic actions *v.* probabilistic actions) produce the following taxonomy of planners:

1. Conditional planning with non-probabilistic actions: These types of planners engage in *conformant* planning: producing a straight-line plan that is guaranteed to succeed no matter what conditions are encountered. Example: CONFORMANT GRAPHPLAN [69].
2. Contingent planning with non-probabilistic actions: Sensing allows this type of planner to produce a contingent plan, but the lack of probabilistic actions means that the planner must look for a plan that will succeed under all circumstances. Examples: CNLP [62], PLINTH [25], SENSORY GRAPHPLAN [72], CASSANDRA [64].
3. Conditional planning with probabilistic actions: As in Case 1, these planners engage in conformant planning, but the probabilities attached to action outcomes allow the planner to specify the straight-line plan that has the highest probability of succeeding, even if that probability is less than 1.0. Example: BURIDAN [41] and UDTPOP [63]. The first planner we developed, MAXPLAN [49], falls into this category.
4. Contingent planning with probabilistic actions: As in case 2, sensing allows planners in this category to produce contingent plans. As in case 3, probabilistic actions allow the

[3] Note that the term *conditional* has been used in different ways in the literature. Plans in which the execution of actions depends on the outcomes of earlier actions were originally called "conditional plans" [71]. Some researchers [19] suggested calling such plans "contingent plans", reserving the term "conditional" for plans in which only the effects of actions are contingent on the outcomes of earlier actions, and this terminology has been generally adopted.

planner to specify the plan that has the highest probability of succeeding. Examples: C-BURIDAN [19], DTPOP [63], MAHINUR [58,59], PGRAPHPLAN/TGRAPHPLAN [7], POMDP:INC_PRUNE [14], HANSEN-FENG [27], GPT [8], and SPUDD [28]. ZANDER, the contingent planner we developed (Section 6), falls into this category, as do traditional OR approaches [4,18,29,65].

Note that cases 2 and 3 subsume case 1, and case 4 subsumes all the other cases; thus, a planner for addressing case 4 can be used in all four scenarios.

Our research has established a novel framework for planning with probabilities based on stochastic satisfiability. In what follows, we will describe the planning-as-satisfiability paradigm and discuss complexity issues that suggest what is necessary to extend the paradigm to probabilistic planning. We will describe the planner we have developed based on this extension, and report results indicating that this is a promising alternative approach to attacking problems in case 4 above.

3. Deterministic planning as satisfiability

Since our work is an extension of the planning-as-satisfiability paradigm for deterministic planning problems, we will describe a representation for such problems, provide a formal definition for the satisfiability problem, show how deterministic planning problems can be encoded as SAT problems, and briefly describe how SATPLAN solves the SAT encoding of a planning problem.

3.1. Representing deterministic planning problems

A planning domain $M = \langle \mathcal{S}, s_0, \mathcal{A}, \mathcal{G} \rangle$ is characterized by a finite set of states $\mathcal{S}$, an initial state $s_0 \in \mathcal{S}$, a finite set of operators or actions $\mathcal{A}$, and a set of goal states $\mathcal{G} \subseteq \mathcal{S}$. The application of an action a in a state s results in a deterministic transition to a new state s'. The objective is to choose actions, one after another, to move from the initial state s_0 to one of the goal states.

The STRIPS representation [21] of M, which we will describe informally, uses a propositional state representation; a state is described by an assignment to a set of Boolean variables. Actions are specified by three sets of propositions:

1. The *preconditions* set specifies what propositions need to be `True` for the action to be executed.
2. The *add effects* set specifies those propositions that become `True` as a result of executing the action, and
3. The *delete effects* set specifies those propositions that become `False` as a result of executing the action.

3.2. Deterministic satisfiability

Informally, a deterministic satisfiability (SAT) problem asks whether a given Boolean formula has a satisfying assignment; that is, is there an assignment of truth values to the variables used in the formula such that the formula evaluates to `True`. SAT is a fundamental problem in computer science. It was the first NP-complete problem and many important, practical problems in areas such as planning and scheduling, network design, and data storage and retrieval (to name just a few) can be expressed as SAT problems [22]. As such, SAT is a very well-studied problem, both from a theoretical point of view (e.g., how does the solution difficulty of random SAT problems vary as one varies the parameters of the problem?) as well as a practical point of view (e.g., how can one solve SAT problems efficiently?).

Formally, let $\mathbf{x} = \langle x_1, x_2, \ldots, x_n \rangle$ be a collection of n Boolean variables, and $\phi(\mathbf{x})$ be a Boolean formula on these variables in conjunctive normal form (CNF) with m clauses. Each clause is a disjunction of *literals*; a literal is a variable or its negation. Thus, ϕ evaluates to `True` if and only if there is at least one literal with the value `True` in every clause. (Note: We will sometimes use 1/0 to denote `True`/`False`.) An *assignment* is a mapping from $\mathbf{x}$ to the set {`True`, `False`}. An assignment A is *satisfying*, and $\phi(\mathbf{x})$ is said to be *satisfied*, if $\phi(\mathbf{x})$ evaluates to `True` under the mapping A. This can be expressed using existential quantifiers and, anticipating the notation necessary for stochastic satisfiability, the expectation of formula satisfaction:

$$\exists x_1, \ldots, \exists x_n \big(E[\phi(\mathbf{x}) \leftrightarrow \texttt{True}] = 1.0 \big).$$

In words, this asks whether there exist values for all the variables such that the probability of the formula evaluating to `True` is certain. Note that we use equivalence ($\leftrightarrow$ `True`) to denote the event of the formula evaluating to `True`.

3.3. Encoding deterministic planning problems as SAT problems

The generality of propositional satisfiability makes it possible to encode deterministic planning problems in a number of different ways; many different approaches to planning can be converted to propositional satisfiability. Both state-space planning and plan-space (causal) planning can be used as a basis for satisfiability encodings [31,53]. For example, one possible SAT encoding of a planning problem is the linear encoding with classical frame axioms [31]. In this type of SAT encoding, satisfiability is made equivalent to goal achievement by enforcing the following conditions:

- the initial conditions and goal conditions hold at the appropriate times (note that the initial state is completely specified whereas the goal state may be only partially specified),
- exactly one action is taken at each time step,
- if an action holds at time t, its preconditions hold at time $t-1$, its add effects hold at time t, and the negation of each of its delete effects holds at time t, and
- if an action does not affect a state variable, then that state variable remains unchanged when that action is executed (classical frame conditions).

The advantages of SAT's expressive generality are clear, but there is also a disadvantage. The multiplicity of possible SAT encodings for a particular problem and the absence of a principled way of selecting the best encoding make it difficult to develop a system that operates as efficiently as possible on a broad range of planning problems. In fact, one of the current challenges in the planning-as-satisfiability paradigm is to automate the process of producing the most efficient SAT encoding of a planning problem [20].

3.4. Solving deterministic satisfiability problems

The most straightforward technique for solving the SAT encoding of the planning problem is *systematic search* for a satisfying assignment. This can perhaps best be visualized by thinking of it as a search on an *assignment tree*. First, impose an arbitrary ordering on the variables. An assignment tree is a binary tree in which each node represents a variable and a partial assignment. The root node at level 0 represents the first variable in the ordering and the empty partial assignment. For node q at level d representing the dth variable v in the variable ordering and partial assignment A, the left child of node q, q_l, represents the variable following v in the variable ordering and the partial assignment A extended by setting v to `True`. The right child of node q, q_r, represents the variable following v in the variable ordering and the partial assignment A extended by setting v to `False`. The 2^n nodes at level n represent all possible complete assignments to the n variables. A traversal of this tree, evaluating the Boolean formula given the full assignment at each leaf, will consider all possible assignments and, hence, is guaranteed to find a satisfying assignment if one exists. The full assignment tree is, of course, exponential in the number of variables, and practical considerations demand that a systematic solver search as little of this tree as possible. We will describe heuristics for this purpose later in this section.

Even using heuristics, however, systematic search is impractical for very large problems. SAT encodings of even moderately-sized planning problems can be very large (> 5000 variables), and for problems of this size a more practical approach is to use *stochastic local search*. SATPLAN, in fact, uses WALKSAT [68], a generic satisfiability algorithm based on stochastic local search. WALKSAT is not complete; it may not find a satisfying assignment when one exists. In addition, it cannot report that a satisfying assignment does not exist (although recent work by Schöning [66] provides probability bounds on the likelihood of missing a satisfying assignment if one exists). WALKSAT, however, can solve satisfiability problems that are orders of magnitude larger than those the best systematic solvers can handle [68].

4. Complexity results

In its most general form, a plan is a *program* that takes as input observable aspects of the environment and produces actions as output. We will classify plans by their *size* (the number of internal states) and *horizon* (the number of actions produced *en route* to a goal state). The computational complexity of propositional planning varies with bounds on the plan size and plan horizon. In the deterministic case, for example, unbounded STRIPS

planning is PSPACE-complete [13]. If we put a polynomial bound on the plan horizon [34], however, STRIPS planning becomes an NP-complete problem.

The complexity of probabilistic propositional planning varies in a similar fashion. If the plan size is unbounded and the plan horizon is infinite, the problem is EXP-complete if states are completely observable [43], or, in the more general case, undecidable [46]. If plan size *or* plan horizon alone is bounded by a polynomial in the size of the representation of the problem, the problem is PSPACE-complete ([44] for plan size and [43] for plan horizon). Contingent planning with polynomial bounds on the plan horizon falls into this class. Evaluating a probabilistic plan—calculating the probability that the given plan reaches a goal state—is PP-complete [44]. Finally, if we place bounds—polynomial in the size of the planning problem—on both plan size and plan horizon, the planning problem is NP^{PP}-complete [44].

The class PP can be informally characterized as the set of problems in which one needs to *count* the number of answers that satisfy some conditions (it is the decision-problem version of #P). PSPACE is the class of problems solvable using polynomial space. Papadimitriou [61] describes these classes in detail.

To the extent that we take the planning problem to be one of constructing a good controller and executing it to solve the problem, polynomial bounds on plan size and plan horizon are reasonable. In some cases, it may not help to know whether a plan exists if that plan is intractable to express, requiring, say, exponential space (and exponential time) to write down. The polynomial bound on plan horizon is perhaps less defensible but nonetheless seems like a reasonable restriction. When a contingent plan is required (see Section 6), the polynomial restriction on plan size may be too severe to allow a good plan (one with a sufficiently high probability of reaching a goal state) to be found, but the polynomial bound on plan horizon is still necessary to keep the problem in a "reasonable" complexity class (PSPACE).

The success of SATPLAN encourages us to try a similar approach for probabilistic planning problems, but these complexity results make it clear that we cannot encode probabilistic planning problems as SAT problems. The relationship among these classes can be summarized as follows:

$$\mathrm{NP} \subseteq \mathrm{PP} \subseteq \mathrm{NP}^{\mathrm{PP}} \subseteq \mathrm{PSPACE}.$$

We currently cannot express an NP^{PP}-complete or PSPACE-complete problem as a compact instance of SAT; if we want to extend the planning-as-satisfiability paradigm to probabilistic planning, we will need a different type of satisfiability problem.

To extend the planning-as-satisfiability paradigm, we need a satisfiability problem that can be used to capture probabilistic planning problems. Stochastic satisfiability, which we describe next, satisfies this requirement.

5. Stochastic satisfiability

Stochastic satisfiability (SSAT) is at the core of the probabilistic planning technique we have developed; ZANDER operates by converting the planning problem to an instance of stochastic satisfiability and solving that problem instead.

Recall the definition of satisfiability from Section 3.2. Given Boolean variables $\mathbf{x} = \langle x_1, x_2, \ldots, x_n \rangle$ and a CNF formula $\phi(\mathbf{x})$ constructed from these variables, the satisfiability problem asks

$$\exists x_1, \ldots, \exists x_n \big(E[\phi(\mathbf{x}) \leftrightarrow \texttt{True}] = 1.0 \big) :$$

Do there exist values for $x_1, x_2, \ldots, x_n$ such that the probability of $\phi(\mathbf{x})$ evaluating to `True` is certain?

The key idea underlying stochastic satisfiability (SSAT) is the introduction of a *randomized quantifier*: ∀. Randomized quantifiers introduce uncertainty into the question of whether there is a satisfying assignment. We will formalize this notion later in this section but, for now, a simple example will illustrate the operation of this quantifier. Suppose we have the following formula:

$$\exists x_1, \forall y_2 \big(E[(x_1 \vee \bar{y}_2) \wedge (\bar{x}_1 \vee y_2) \leftrightarrow \texttt{True}] \geqslant 0.75 \big). \tag{1}$$

This instance of SSAT asks whether a value for x_1 can be chosen such that *for random* values of y_2 (choose `True` or `False` with equal probability) the *expected* probability of satisfying the indicated Boolean formula is at least 0.75. This extension of SAT was first explored by [60].

There are two important points to be made here. First, the presence of randomized quantifiers means that obtaining a satisfying assignment is no longer completely under the control of the solver. In the above example, after the solver has chosen a value for the existentially quantified variable x_1, the value of the randomly quantified variable y_2 will be chosen by flipping a fair coin. Thus, there is a certain *probability* that the choice of a value for x_1 will lead to a satisfied formula. If the solver sets x_1 to `True`, then there is a 0.5 probability that the formula will be satisfied (if the coin flip for y_2 comes up `True`) and a 0.5 probability that the formula will be unsatisfied (if the coin flip comes up `False`). The situation is similar if the solver sets x_1 to `False`. (Since the solver can choose values for the existentially quantified variables and the probability of satisfaction depends on the chance outcomes of the randomized variables, we will sometimes refer to existentially quantified variables as *choice variables* and randomly quantified variables as *chance variables*.)

Second, quantifier ordering is now critical. In the example, a value for x_1 must be chosen that yields a sufficiently high probability of satisfaction regardless of the randomly chosen value for y_2. This is impossible; either value of x_1 will result in an unsatisfied formula for one of y_2's values, so the maximum probability of satisfaction is 0.5. Suppose, however, the order of the quantifiers were reversed:

$$\forall y_1, \exists x_2 \big(E[(x_2 \vee \bar{y}_1) \wedge (\bar{x}_2 \vee y_1) \leftrightarrow \texttt{True}] \geqslant 0.75 \big).$$

Here, the choice of a value for x_2 can be made *contingent* on the random outcome of the coin flip establishing y_1's value. In this case, choosing x_2's value to be the same as y_1's value leads to a satisfied formula regardless of the coin flip. The probability of satisfaction is now 1.0, exceeding the specified threshold.

Formally, an SSAT formula is defined by a triple (ϕ, Q, θ) where ϕ is a CNF formula with underlying ordered variables $x_1, \ldots, x_n$, Q is a mapping from variables to quantifiers (existential ∃ and randomized ∀), and $0 \leqslant \theta \leqslant 1$ is a satisfaction threshold. Define $\phi\lceil_{x_i=b}$

to be the $(n-1)$-variable CNF formula obtained by assigning the single variable x_i the Boolean value b in the n-variable CNF formula ϕ and simplifying the result, including any necessary variable renumbering. (Variables are numbered so that x_1 corresponds to the outermost, or leftmost, quantifier and x_n to the innermost.)

The maximum probability of satisfaction, or value, of ϕ (under quantifier order Q), $val(\phi, Q)$, is defined by induction on the number of quantifiers. Let x_1 be the variable associated with the outermost quantifier. Then:

1. if ϕ contains an empty clause, then $val(\phi, Q) = 0.0$;
2. if ϕ contains no clauses then $val(\phi, Q) = 1.0$;
3. if $Q(x_1) = \exists$, then $val(\phi, Q) = \max(val(\phi\lceil_{x_1=0}, Q), val(\phi\lceil_{x_1=1}, Q))$;
4. if $Q(x_1) = \forall$, then $val(\phi, Q) = (val(\phi\lceil_{x_1=0}, Q) + val(\phi\lceil_{x_1=1}, Q))/2$.

Given ϕ, Q, and a threshold θ, (ϕ, Q, θ) is `True` if and only if $val(\phi, Q) \geqslant \theta$.

Let us examine the application of this definition to the original example (Eq. 1). The outermost quantifier is existential, so Rule 3 dictates that the value of the formula is the maximum of the value of the subformula if x_1 is `True` and the value of the subformula if x_1 is `False`. If x_1 is `True`, the formula reduces to $\forall y_1(E[(y_1) \leftrightarrow \texttt{True}] \geqslant 0.75)$ (after variable renumbering). Since the outermost quantifier is now randomized, Rule 4 dictates that the value of this subformula is the average of the values if y_1 is `True` and if y_1 is `False`. If y_1 is `True`, the new subformula contains no clauses and the value is 1.0 (Rule 2). If y_1 is `False`, the new subformula contains an empty clause and the value is 0.0 (Rule 1). The average of these, 0.5, is thus the value of the subformula when x_1 is `True`. If x_1 is `False`, a similar calculation establishes the value of the subformula to be 0.5. Taking the maximum, the value of the original formula is 0.5. Since the threshold θ is 0.75, the SSAT instance $(\phi = (x_1 \vee \bar{y}_2) \wedge (\bar{x}_1 \vee y_2)$, $Q = \{(x_1, \exists), (y_2, \forall)\}$, $\theta = 0.75)$ is `False`.

One further modification is necessary to encode planning problems as stochastic satisfiability problems. We will allow an arbitrary, rational probability to be attached to a randomly quantified variable. This probability will specify the likelihood with which that variable will have the value `True`. Thus, the value of a randomly quantified variable will be determined according to this probability, rather than choosing `True` or `False` with equal probability. This has an impact both on notation and on the inductive definition of value. Randomized quantifiers can now be superscripted with an associated probability other than 0.5. For example, $\forall^{0.65} y_1$ indicates that the chance variable y_1 is `True` with probability 0.65. Rule 4 in the inductive definition of $val(\phi, Q)$ becomes:

4. if $Q(x_1) = \forall^{\pi}$, then
   $val(\phi, Q) = (val(\phi\lceil_{x_1=0}, Q) \times (1.0 - \pi) + val(\phi\lceil_{x_1=1}, Q) \times \pi)$.

In other words, the value in this case is the probability weighted average of the values of the two possible subformulas.

For the sake of completeness, we note here that stochastic satisfiability can be extended to include universally quantified variables as well as existentially and randomly quantified variables. Although this version of stochastic satisfiability might be useful for encoding planning problems when there is an adversarial influence, we do not use this version in any

of our SSAT-based planners. Details regarding this *Extended* SSAT problem are available elsewhere [45].

5.1. Solving stochastic satisfiability problems

We describe `evalssat`, a sound and complete algorithm for solving SSAT problems. Given an arbitrary SSAT instance (ϕ, θ, Q), this algorithm is guaranteed to return the correct answer, although the running time can be exponential. The `evalssat` algorithm can be viewed as an extension of the Davis–Putnam–Logemann–Loveland (DPLL) algorithm for solving SAT problems [16]. To our knowledge, DPLL and its variants are the best systematic satisfiability solvers known. As such (and also because of its simplicity), DPLL was the obvious choice as a basis for an SSAT solver. DPLL works by enumerating all possible assignments, simplifying the formula whenever possible. These simplifications, or pruning rules, make it possible to solve problems whose entire set of assignments could not be completely enumerated. Since DPLL is designed to solve SAT problems, the pruning rules only need to deal with existential quantifiers. The `evalssat` algorithm extends the DPLL algorithm to SSAT by providing pruning rules for randomized quantifiers.

The `evalssat` algorithm (Fig. 1) takes formula ϕ, quantifier order Q, and low and high thresholds θ_l and θ_h. It returns a value less than θ_l if and only if the value of the SSAT formula is less than θ_l, a value greater than θ_h if and only if the value of the SSAT formula is greater than θ_h, and otherwise the exact value of the SSAT formula. (Note that π_b^v denotes the probability that randomized variable v has value b.) Thus, this algorithm can be used to solve the SSAT decision problem by setting $\theta_l = \theta_h = \theta$. It can also be used to compute the exact value of the formula by setting $\theta_l = 0$ and $\theta_h = 1$. The algorithm's basic structure is to compute the value of the SSAT formula from its definition (Section 5); this takes place in the first two lines of pseudocode and in the section of pseudocode labeled "*Splitting*", which enumerates all assignments, applying operators recursively from left to right. However, it is made more complex (and efficient) by a set of pruning rules, described next.

5.1.1. Unit propagation

When a Boolean formula ϕ is evaluated that contains a variable x_i that appears alone in a clause in ϕ with sign b (0 if $\bar{x_i}$ is in the clause, 1 if x_i is in the clause), the normal left-to-right evaluation of quantifiers can be interrupted to deal with this variable. This is called *unit propagation* and x_i is referred to as a *unit variable*, by analogy with DPLL.

If the quantifier associated with x_i is existential, x_i can be eliminated from the formula by assigning it value b and recurring. As in DPLL, this is valid because assigning $x_i = 1 - b$ is guaranteed to make ϕ `False`, and $x_i = b$ can be no worse. Similarly, if the quantifier associated with x_i is randomized, it is the case that one branch of the computation will return a zero, so x_i can be eliminated from the formula by assigning it value b and continuing recursively. The resulting value is multiplied by the probability associated with the forced value of the randomized quantifier ($\pi_b^{x_i}$), since it represents the value of only one branch.

```
evalssat(φ, Q, θ_l, θ_h) := {
  if φ is the empty set, return 1.0
  if φ contains an empty clause, return 0.0
  /* Unit Propagation */
  if x_i is a unit variable with sign b and Q(x_i) = ∃,
    return evalssat(φ⌈_{x_i=b}, Q, θ_l, θ_h)
  if x_i is a unit variable with sign b and Q(x_i) = ∀,
    return evalssat(φ⌈_{x_i=b}, Q, θ_l/π_b^{x_i}, θ_h/π_b^{x_i})π_b^{x_i}
  /* Pure Variable Elimination */
  if x_i is a pure variable with sign b and Q(x_i) = ∃,
    return evalssat(φ⌈_{x_i=b}, Q, θ_l, θ_h)
  /* Splitting */
  if Q(x_1) = ∃, {
    v_0 = evalssat(φ⌈_{x_1=0}, Q, θ_l, θ_h)
    if v_0 ⩾ θ_h, return v_0
    v_1 = evalssat(φ⌈_{x_1=1}, Q, max(θ_l, v_0), θ_h)
    return max(v_0, v_1)
  }
  if Q(x_1) = ∀, {
    v_0 = evalssat(φ⌈_{x_1=0}, Q, (θ_l − π_1^{x_1})/π_0^{x_1}, θ_h/π_0^{x_1})
    if v_0π_0^{x_1} + π_1^{x_1} < θ_l, return v_0π_0^{x_1}
    if v_0π_0^{x_1} ⩾ θ_h, return v_0π_0^{x_1}
    v_1 = evalssat(φ⌈_{x_1=1}, Q, (θ_l − v_0π_0^{x_1})/π_1^{x_1}, (θ_h − v_0π_0^{x_1})/π_1^{x_1})
    return v_0π_0^{x_1} + v_1π_1^{x_1}
  }
}
```

Fig. 1. The `evalssat` algorithm generalizes the DPLL algorithm for satisfiability to solve SSAT problems. Note: π_b^v denotes the probability that randomized variable v has value b.

5.1.2. Pure variable elimination

Pure variable elimination applies when there is a pure variable; i.e., a variable x_i that appears only with one sign b in ϕ. If $Q(x_i) = \exists$, the algorithm assigns $x_i = b$ and recurs. This is valid because there are no unsatisfied clauses that would be satisfied if $x_i = 1 - b$ but unsatisfied if $x_i = b$. Interestingly, pure variable elimination does not appear to be possible for randomized variables. Both assignments to a randomized variable give *some* contribution to the value of the SSAT formula, and must be considered independently.[4]

5.1.3. Threshold pruning

Another useful class of pruning rules concerns the *threshold* parameters θ_l and θ_h. While some care must be taken to pass meaningful thresholds when applying unit propagation, threshold pruning mainly comes into play when variables are split to try to prevent recursively computing both assignments to x_1, the outermost quantified variable. Note that threshold pruning is similar to MINIMAX tree **cutoffs* [1].

[4] In fact, pure variable elimination complicated our implementation and did not appear to provide a significant improvement. We did not use this optimization in our experimental results.

If $Q(x_1) = \exists$, after the first recursive call computing v_0 (the value of the current formula with x_1 set to `False`), it is possible that θ_h has already been exceeded. In this case, the algorithm can simply return v_0, without ever computing v_1 (the value of the current formula with x_1 set to `True`). In particular, it is possible that $v_1 > v_0$, but all that is significant is whether one of the two exceeds θ_h. If v_0 exceeds θ_l but falls short of θ_h, this can be used to increase the lower threshold for the recursive computation of v_1; since the algorithm must take the larger of v_0 and v_1, the precise value of v_1 is not needed if it less than v_0.

Threshold pruning is not as strong for randomized variables, although it can be done. There are two types of threshold pruning that apply. First, if the value obtained by assigning 0 to x_1 (v_0) is so low that, even if the value obtained by assigning 1 to x_1 (v_1) attains its maximum value of 1.0, the low threshold will not be met ($v_0\pi_0^{x_1} + \pi_1^{x_1} < \theta_l$), then the algorithm can return $v_0\pi_0^{x_1}$ without calculating v_1. Second, if v_0 is high enough to meet the high threshold even if $v_1 = 0.0$ ($v_0\pi_0^{x_1} \geqslant \theta_h$), the algorithm can, again, return $v_0\pi_0^{x_1}$ without computing v_1. If both tests fail, the algorithm needs to compute v_1, but can adjust the thresholds accordingly.

The opportunities for threshold pruning are greatest when $\theta_l = \theta_h$, but, in this case, the `evalssat` algorithm may not return the optimum probability of satisfaction. Empirical tests, however, indicate that there can be significant—although possibly diminished—benefits from threshold pruning even when θ_l is set to 0.0 and θ_h is set to 1.0, thus forcing the `evalssat` algorithm to find the optimum probability of satisfaction while still applying threshold pruning wherever possible internally.

For a detailed explanation of thresholds, see the proof of correctness of `evalssat` by Littman et al. [45].

6. Contingent planning

When planning under uncertainty, any information about the state of the world is precious. A *contingent plan* is one that can make action choices contingent on such information. In this section, we will describe the sequential-effects-tree representation (ST) [43], the propositional representation we use for probabilistic contingent planning problems, and provide an example to illustrate the representation.

6.1. Representing probabilistic contingent planning problems

ZANDER, which we will describe in Section 7, works on partially observable probabilistic propositional planning domains. Recall from Section 3.1 that a deterministic planning domain $M = \langle \mathcal{S}, s_0, \mathcal{A}, \mathcal{G} \rangle$ is characterized by a finite set of states $\mathcal{S}$, an initial state $s_0 \in \mathcal{S}$, a finite set of operators or actions $\mathcal{A}$, and a set of goal states $\mathcal{G} \subseteq \mathcal{S}$. Nearly the same tuple characterizes probabilistic planning problems, except that now the initial state is a probability distribution over states (i.e., the initial state is uncertain), the application of an action a in a state s results in a *probabilistic* transition to a new state s', and an observation function is needed to specify how states are "perceived" to allow for contingent planning. The objective is to choose actions, one after another, to move from the initial probability

distribution to a probability distribution in which the probability of being in a goal state is greater than or equal to some threshold θ in a fixed number of steps.[5]

We use a propositional representation called the sequential-effects-tree representation (ST) [43], which is a syntactic variant of two-time-slice Baye's nets (2TBNs) with conditional probability tables represented as trees [9,11]. (This representation is also equivalent to the STRIPS-like probabilistic state-space operators, or PSOs, [26,43].)

The ST representation of a planning domain can be defined formally as $\mathbb{M} = \langle \mathbb{P}, \mathbb{I}, \mathcal{A}, \mathbb{T}, \mathbb{G}_{\mathsf{T}}, \mathbb{G}_{\mathsf{F}}, \mathbb{O} \rangle$. Here, $\mathbb{P}$ is a finite set of n distinct propositions. The set of states is the power set of $\mathbb{P}$; the propositions in state s are said to be "true" in s.

The transition function is represented by a function $\mathbb{T}$, which maps each action in $\mathcal{A}$ to an ordered sequence of binary decision trees. Each of these decision trees has a label proposition, decision propositions at the nodes (optionally labeled with the suffix "**:new**"), and probabilities at the leaves. The decision trees $\mathbb{T}(a)$ for action a define the transition probability from state s to state s' as follows. For each decision tree i, let $\mathbf{p}_i$ be its label proposition. Define ρ_i to be the value of the leaf node found by traversing decision tree $\mathbb{T}(a)_i$, taking the left branch if the decision proposition is in s (or s' if the decision proposition has the "**:new**" suffix) and the right branch otherwise. Finally, we define the transition probability to be

$$\prod_i \begin{cases} \rho_i, & \text{if } \mathbf{p}_i \in s', \\ 1-\rho_i, & \text{otherwise.} \end{cases}$$

This definition ensures a well-defined probability distribution over s' for each a and s. Using decision trees for next-state distributions captures *variable independence* (independence among variables regardless of their values) as well as *propositional independence* (independence of specific variable assignments) [9].

To ensure the validity of the representation, we only allow "**p:new**" to appear as a decision proposition in $\mathbb{T}(a)_i$ if $\mathbf{p}$ is a label proposition for some decision tree $\mathbb{T}(a)_j$ for $j < i$. For this reason, the order of the decision trees in $\mathbb{T}(a)$ is significant. This is analogous to the requirement of acyclicity in belief networks.

The initial state $\mathbb{I}$ can be thought of as a special transition from a state s_{init} in which all propositions are `False` (the actual truth values are immaterial) via a mandatory "set-up" action $a_{\text{set-up}}$ that establishes the actual initial state for a particular instance of the planning problem. Note that any propositions appearing in the decision trees for $a_{\text{set-up}}$ must have the suffix **:new**, as there is no previous state to refer to.

The sets $\mathbb{G}_{\mathsf{T}}$ and $\mathbb{G}_{\mathsf{F}}$ are the sets of propositions that are, respectively, `True` and `False` in a goal state, so the set of goal states $\mathcal{G}$ is the set of states s such that $\mathbb{G}_{\mathsf{T}} \subseteq s$ and $\mathbb{G}_{\mathsf{F}} \subseteq \mathbb{P} - s$.

To represent contingent planning problems, the original ST representation is augmented by declaring a subset of the state propositions $\mathbb{O} \subseteq \mathbb{P}$ to be *observable propositions*. These are the propositions on which the agent's action decisions can be conditioned.

[5] This is just one possibility. Another commonly used objective is that of maximizing expected discounted reward [10,11,39,40]. A planning problem with this objective can be transformed into an equivalent goal-oriented probabilistic planning problem [15,74]. See Appendix A for a proof.

This observation model is a natural generalization of the observation functions used in POMDPs—it is straightforward to emulate the POMDP representation via observable propositions, as we show in our example in the next section. Because the truth values of observable propositions can be set probabilistically, a domain designer can make a domain fully observable, unobservable, or have observations depend on actions and states in probabilistic ways.

The planning task is to find a plan that selects an action for each step t as a function of the value of observable propositions for steps before t. We want to find a plan that exceeds a user-specified threshold for the probability of reaching a goal state in a fixed number of steps, if one exists. An alternate formulation is to maximize the probability of reaching a goal state.

6.2. Example domain

Consider a simple domain based on the TIGER problem [30]. The domain consists of four propositions: **tiger-behind-left-door**, **dead**, **rewarded** and **hear-tiger-behind-left-door**, the last of which is observable. In the initial state, **tiger-behind-left-door** is `True` with probability 0.5, **dead** is `False`, **rewarded** is `False`, and **hear-tiger-behind-left-door** is irrelevant. The goal states are specified by the partial assignment (**rewarded**, (not **dead**)). The three actions are listen-for-tiger, open-left-door, and open-right-door (Fig. 2). Actions open-left-door and open-right-door make **rewarded** `True`, as long as the tiger is

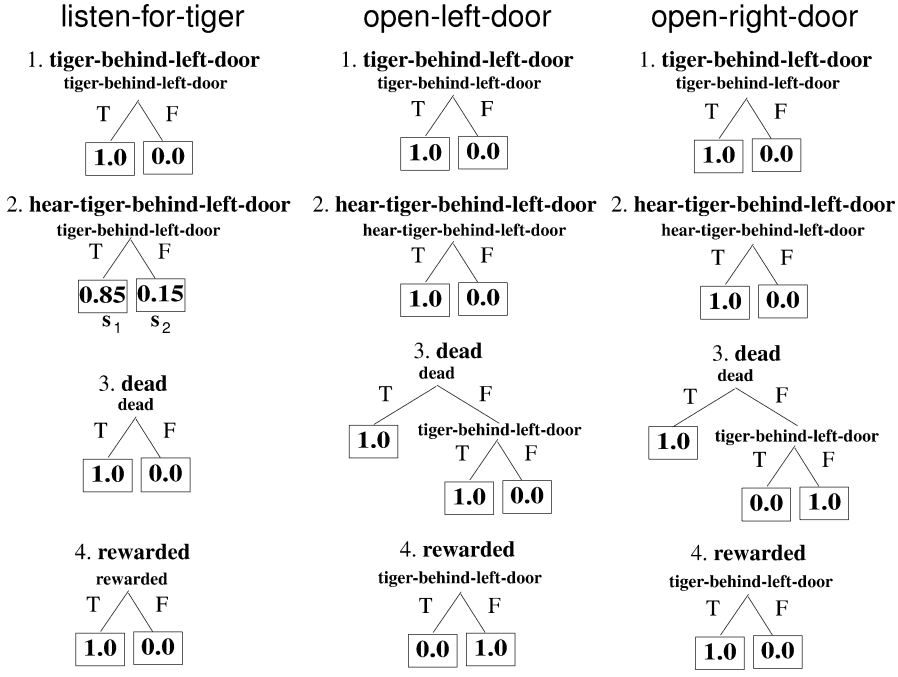


Fig. 2. The effects of the actions in the TIGER problem are represented by a set of decision trees.

not behind that door (we assume the tiger is behind the right door if **tiger-behind-left-door** is `False`). Since **tiger-behind-left-door** is not observable, the listen action becomes important; it causes the observable **hear-tiger-behind-left-door** proposition to become equal to **tiger-behind-left-door** with probability 0.85 (and its negation otherwise). By listening multiple times, it becomes possible to determine the likely location of the tiger.

As the tiger problem was originally specified as a POMDP, this example illustrates how a POMDP representation is converted to ST. First, the reward for opening the correct door is captured by transitions to the goal state. Similarly, the punishment for opening the incorrect door is captured by a transition to a state in which **dead** is `True`, eliminating the possibility of future goal achievement. To emulate a slight cost for listening, a low probability of making **dead** `True` could have been added to the description of the listen-for-tiger action.

In the original problem, the listen-for-tiger action has an associated observation function. This idea is captured directly through the observable **hear-tiger-behind-left-door** proposition—the observation probabilities map exactly to the probability that the observable proposition is `True` after the action.

This particular example does not make use of **:new** suffix, as propositions are independent functions of the previous state.

6.3. Probabilistic planning language

Although the ST representation is the formal representation language underlying ZANDER, for convenience we write down this representation using the Probabilistic Planning Language ($\mathcal{PPL}$). $\mathcal{PPL}$ is a high-level action language that extends the action language $\mathcal{AR}$ [23] to support probabilistic domains. An ST representation can be easily expressed by $\mathcal{PPL}$. Each path through each decision tree is replaced by a $\mathcal{PPL}$ statement. The general form of a $\mathcal{PPL}$ statement for a path through the decision tree describing action a's impact on proposition p is:

a causes p withp π if c_1 and c_2 and $\ldots$ and c_m,

where $0.0 \leqslant \pi \leqslant 1.0$ is the probability at the leaf, and $c_i, 1 \leqslant i \leqslant m$, are the state propositions described by the particular path. In words, the statement says that if conditions $c_i, 1 \leqslant i \leqslant m$, are `True` when action a is executed, p will become `True` with probability π. For example, the left path in the decision tree describing listen-for-tiger's effect on **hear-tiger-behind-left-door** (Fig. 2) would be expressed in the following $\mathcal{PPL}$ statement:

listen-for-tiger causes **hear-tiger-behind-left-door** withp 0.85

if **tiger-behind-left-door**.

In addition to providing a convenient way of writing down ST decision trees, $\mathcal{PPL}$ gives users the (optional) opportunity to easily express state invariants, equivalences, irreversible conditions, and action preconditions—information that can greatly decrease the time required for the SSAT solver to find a solution.

7. ZANDER

In this section, we present ZANDER, an implemented framework for contingent planning under uncertainty using stochastic satisfiability. ZANDER is based on MAXPLAN, a noncontingent planner we developed earlier [49].

7.1. Encoding contingent planning problems as SSAT problems

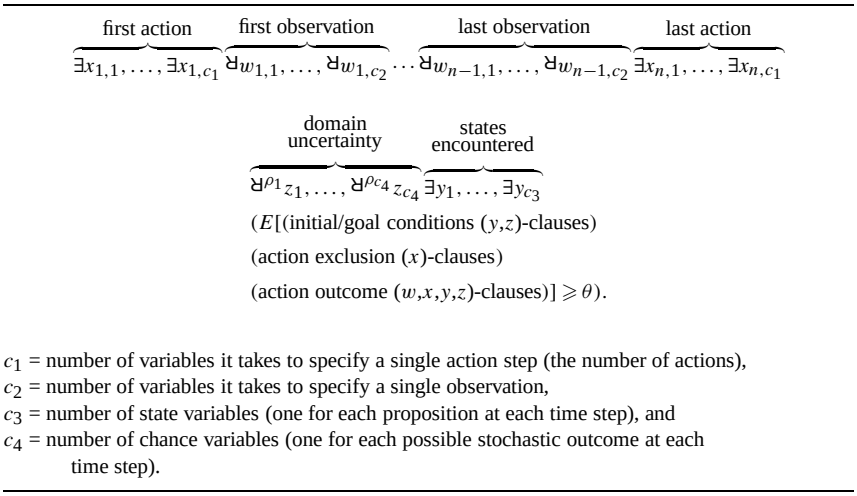
The problem conversion unit of ZANDER is a Java program that takes as input an ST representation of a planning problem expressed in $\mathcal{PPL}$ and converts it into an SSAT formula. As with SATPLAN, the conversion process requires that the number of steps in the plan be chosen in advance. Searching for the appropriate plan length is external to the encoding process.

In Section 7.1.1, we discuss how the ordering of quantified variables is used to encode a contingent planning problem. In Sections 7.1.2 and 7.1.3, we describe in detail how the clauses in the SSAT problem are generated from the ST representation of the planning problem.

7.1.1. Quantifier ordering

In an SSAT formula, the value of an existential variable x can be selected on the basis of the values of all the variables to x's left in the quantifier sequence. Thus, viewing an existential variable as an action choice, the values of all "earlier" variables in the quantifier sequence are observable at the time x's value is selected. So, the choice represented by x is contingent on the earlier variables. This allows one to map contingent planning problems to stochastic satisfiability by encoding the contingent plan in the decision tree induced by the quantifier ordering associated with the SSAT formula. By alternating blocks of existential variables that encode actions and blocks of randomized variables that encode observations, one can condition the value chosen for any action variable on the possible values for all the observation variables that appear earlier in the ordering. A generic SSAT encoding for contingent plans appears in Fig. 3. Note that this approach is agnostic as to the structure of the plan; the type of plan returned is algorithm dependent. ZANDER solves an SSAT instance by constructing a tree-structured proof; this corresponds to generating tree-structured plans that contain a branch for each observable variable. Other SSAT solvers could produce DAG-structured, subroutine-structured, or value-function-based plans, depending on how they attack SSAT problems.

The quantifiers naturally fall into three segments: a plan-execution history, the domain uncertainty, and the result of the plan-execution history given the domain uncertainty. The plan-execution-history segment is an alternating sequence of choice-variable blocks (one for each action choice) and chance-variable blocks (one for each set of possible observations at a time step). This segment begins with the action-variable block for the first (non-contingent) action choice and ends with the action-variable block for the last action choice. The action choice encoded in each action-variable block can, thus, be conditioned on the values of all the preceding observation variables in all the observation-variable blocks to the left of that action-variable block. In the TIGER problem, each action-variable block would be composed of the three possible actions—listen-for-tiger, open-left-door, and



c_1 = number of variables it takes to specify a single action step (the number of actions),
c_2 = number of variables it takes to specify a single observation,
c_3 = number of state variables (one for each proposition at each time step), and
c_4 = number of chance variables (one for each possible stochastic outcome at each time step).

Fig. 3. A generic SSAT encoding of a contingent planning problem.

open-right-door—and each observation-variable block would be composed of the single variable **hear-tiger-behind-left-door**. This means that the values of the variables in the second action-variable block (i.e., the action chosen) can be conditioned on the value of **hear-tiger-behind-left-door** in the observation-variable block immediately preceding them; i.e., the planner can specify one action if the tiger is heard behind the left door, and a different action otherwise.

The domain uncertainty segment is a single block containing all the chance variables that modulate the impact of the actions on the observation and state variables. These variables are associated with randomized quantifiers; when the algorithm considers a variable that represents uncertainty in the environment, it needs to take the probability weighted average of the success probabilities associated with the two possible settings of the variable. In the TIGER problem, there would be a chance variable (probability = 0.85) associated with the outcome of each listen-for-tiger action.

The result segment is a single block containing all the non-observation state variables. These variables are associated with existential quantifiers, indicating that the algorithm can choose the best truth setting for each variable. In reality, all such "choices" are forced by the settings of the action variables in the first segment and the chance variables in the second segment. If these forced choices are compatible, then the preceding plan-execution history is possible and has a non-zero probability of achieving the goals. Otherwise, either the plan-execution history is impossible, given the effects of the actions, or it has a zero probability of achieving the goals.

Let ϕ and Q represent an SSAT encoding of a contingent planning problem. The probability of satisfaction, or value, $val(\phi, Q)$ is defined by induction on the number of quantifiers, and is similar to the value of an SSAT formula defined in Section 5. Let x_1 be the variable associated with the outermost quantifier. Then:

1. if ϕ contains an empty clause, then $val(\phi, Q) = 0.0$;

2. if ϕ contains no clauses then $val(\phi, Q) = 1.0$;
3. if $Q(x_1) = \exists$, then $val(\phi, Q) = \max(val(\phi\lceil_{x_1=0}, Q), val(\phi\lceil_{x_1=1}, Q))$;
4. if $Q(x_1) = \text{ᗡ}^{\pi}$ and x_1 is not an observation variable, then $val(\phi, Q) = (val(\phi\lceil_{x_1=0}, Q) \times (1.0 - \pi) + val(\phi\lceil_{x_1=1}, Q) \times \pi)$;
5. if $Q(x_1) = \text{ᗡ}^{0.5}$ and x_1 is an observation variable, then $val(\phi, Q) = val(\phi\lceil_{x_1=0}, Q) + val(\phi\lceil_{x_1=1}, Q)$.

The only difference between these rules and those stated in Section 5 for a general SSAT formula is the addition of Rule 5 to handle chance (randomized) variables encoding observations. This rule states that the value of a formula whose outermost quantifier is a chance variable encoding an observation is the *sum* of the value of the formula if that variable is assigned the value `True` and the value of the formula if that variable is assigned the value `False`, rather than the probability weighted average of these two values (as in Rule 4, for the value of a formula whose outermost quantifier is a chance variable that does *not* encode an observation). This special treatment of some chance variables requires some explanation.

The chance variables representing observations in the plan-execution history are used only to mark possible branch points in the plan, and not to encode the probability of actually making that observation. (The actual probability of the observation being `True` is encoded by a chance variable that appears in the domain uncertainty segment.) For example, in the 2-step TIGER problem, there is a choice-variable block representing a choice between actions listen-for-tiger, open-left-door, and open-right-door at time step 1, followed by a single observation chance variable **hear-tiger-behind-left-door**, followed by another choice-variable block, representing a choice between actions listen-for-tiger, open-left-door, and open-right-door at time step 2. The function of chance variable **hear-tiger-behind-left-door** is to allow the solver to choose one action at time step 2 if **hear-tiger-behind-left-door** is `True` and a different action if **hear-tiger-behind-left-door** is `False`.

In order to calculate the correct probability of success of such a branching plan, the algorithm needs to sum the success probabilities over all branches. Making **hear-tiger-behind-left-door** a chance variable (instead of a choice variable) allows one to combine the success probabilities of the two branches, but, as defined for a standard SSAT problem (Rule 4 above), chance variables must combine the success probabilities associated with their two values (`True`/`False`) by taking the probability weighted average of these success probabilities, instead of the sum. To simulate Rule 5 within the SSAT framework, we would associate a probability of 0.5 with the chance variable **hear-tiger-behind-left-door** and adjust the calculated probability of success upward by a factor of 2. This would be equivalent to the more straightforward approach actually used (summing the success probabilities of the two branches).

In the next two sections, we illustrate the variable and clause production process by describing the construction of the CNF formula corresponding to a one-step plan for the TIGER domain.

7.1.2. Variables

The converter first creates a set of propositions that capture the uncertainty in the domain. For each decision-tree leaf l labeled with a probability π_l that is strictly between 0.0 and 1.0, the converter creates a *random proposition* $\mathbf{r}_l$ that is true with probability π_l. For example, in the second decision tree of the listen-for-tiger action (Fig. 2), $\mathbf{s}_1$ is a random proposition that is `True` with probability 0.85. The leaf l is then replaced with a node labeled $\mathbf{r}_l$ having a left leaf of 1.0 and a right leaf of 0.0. This has the effect of slightly increasing the size of decision trees and the number of propositions, but also of simplifying the decision trees so that all leaves are labeled with either 0.0 or 1.0 probabilities.

The converter is given a plan horizon T and time-indexes each proposition and action so the planner can reason about what happens when. Variables are created to record the status of actions and propositions in a T-step plan by taking three separate cross products: actions and time steps 1 through T, propositions and time steps 0 through T, and random propositions and time steps 1 through T. The total number of variables in the CNF formula is

$$V = (A + P + R)T + P,$$

where A, P, and R are the number of actions, propositions, and random propositions, respectively.

The variables generated by the actions are the choice variables. In our example, these are the variables listen-for-tiger-1, open-left-door-1, and open-right-door-1. The variables generated by the random propositions are the chance variables. In our example, we have two random propositions ($\mathbf{s}_1$ and $\mathbf{s}_2$ and the variables generated are $\mathbf{s}_1$-1 and $\mathbf{s}_2$-1). (We will describe the generation and use of these chance variables in more detail later in this section.)

The variables generated by the propositions for time steps 1 through T are choice variables. In the TIGER domain, these choice variables are **tiger-behind-left-door**-1, **hear-tiger-behind-left-door**-1, **dead**-1, and **rewarded**-1. These variables are encoded as "choice" variables, but the choice is forced, given a choice of values for the action variables and an instantiation of values for the chance variables encoding the domain uncertainty. Variables generated by the propositions at time step 0 are either choice variables, if their status is deterministically specified in the initial conditions, or chance variables, if their status is probabilistically specified in the initial conditions.

Each variable indicates the status of an action, proposition, or decision-tree leaf node at a particular time step. So, for example, the variable open-left-door-1, if `True`, indicates that the open-left-door action was taken at time step 1, and the variable s_1-1, if `True`, indicates that the decision-tree leaf node associated with s_1 is `True` at time step 1.

7.1.3. Clauses

The SSAT encoding of the planning problem is constructed to enforce the following conditions:

1. the initial conditions hold at time step 0 and the goal conditions at time step T,
2. actions at time step t are mutually exclusive ($1 \leqslant t \leqslant T$),

Initial Conditions:
1. ($\overline{\textbf{dead}\text{-}0}$) $\wedge$
2. ($\overline{\textbf{rewarded}\text{-}0}$) $\wedge$

Goal Conditions:
3. ($\overline{\textbf{dead}\text{-}1}$) $\wedge$
4. (**rewarded**-1) $\wedge$

Exactly One Action Per Time Step:
5. (listen-for-tiger-1 $\vee$ open-left-door-1 $\vee$ open-right-door-1) $\wedge$

Action Effects:
6. ($\overline{\text{listen-for-tiger-1}} \vee \overline{\textbf{tiger-behind-left-door}\text{-}0} \vee \overline{\textbf{s}_1\text{-}1} \vee \textbf{hear-tiger-behind-left-door}\text{-}1$) $\wedge$
7. ($\overline{\text{listen-for-tiger-1}} \vee \overline{\textbf{tiger-behind-left-door}\text{-}0} \vee \textbf{s}_1\text{-}1 \vee \overline{\textbf{hear-tiger-behind-left-door}\text{-}1}$) $\wedge$
8. ($\overline{\text{listen-for-tiger-1}} \vee \textbf{tiger-behind-left-door}\text{-}0 \vee \overline{\textbf{s}_2\text{-}1} \vee \textbf{hear-tiger-behind-left-door}\text{-}1$) $\wedge$
9. ($\overline{\text{listen-for-tiger-1}} \vee \textbf{tiger-behind-left-door}\text{-}0 \vee \textbf{s}_2\text{-}1 \vee \overline{\textbf{hear-tiger-behind-left-door}\text{-}1}$) $\wedge$
10. ($\overline{\text{open-left-door-1}} \vee \overline{\textbf{hear-tiger-behind-left-door}\text{-}1}$) $\wedge$
11. ($\overline{\text{open-right-door-1}} \vee \overline{\textbf{hear-tiger-behind-left-door}\text{-}1}$) $\wedge$
12. ($\overline{\text{listen-for-tiger-1}} \vee \overline{\textbf{tiger-behind-left-door}\text{-}0} \vee \textbf{tiger-behind-left-door}\text{-}1$) $\wedge$
13. ($\overline{\text{listen-for-tiger-1}} \vee \textbf{tiger-behind-left-door}\text{-}0 \vee \overline{\textbf{tiger-behind-left-door}\text{-}1}$ $\wedge$
14. ($\overline{\text{open-left-door-1}} \vee \overline{\textbf{tiger-behind-left-door}\text{-}0} \vee \textbf{tiger-behind-left-door}\text{-}1$) $\wedge$
15. ($\overline{\text{open-left-door-1}} \vee \textbf{tiger-behind-left-door}\text{-}0 \vee \overline{\textbf{tiger-behind-left-door}\text{-}1}$) $\wedge$
16. ($\overline{\text{open-right-door-1}} \vee \overline{\textbf{tiger-behind-left-door}\text{-}0} \vee \textbf{tiger-behind-left-door}\text{-}1$) $\wedge$
17. ($\overline{\text{open-right-door-1}} \vee \textbf{tiger-behind-left-door}\text{-}0 \vee \overline{\textbf{tiger-behind-left-door}\text{-}1}$) $\wedge$
18. ($\overline{\text{listen-for-tiger-1}} \vee \overline{\textbf{dead}\text{-}0} \vee \textbf{dead}\text{-}1$) $\wedge$
19. ($\overline{\text{listen-for-tiger-1}} \vee \textbf{dead}\text{-}0 \vee \overline{\textbf{dead}\text{-}1}$) $\wedge$
20. ($\overline{\text{open-left-door-1}} \vee \overline{\textbf{dead}\text{-}0} \vee \textbf{dead}\text{-}1$) $\wedge$
21. ($\overline{\text{open-left-door-1}} \vee \textbf{dead}\text{-}0 \vee \overline{\textbf{tiger-behind-left-door}\text{-}0} \vee \textbf{dead}\text{-}1$) $\wedge$
22. ($\overline{\text{open-left-door-1}} \vee \textbf{dead}\text{-}0 \vee \textbf{tiger-behind-left-door}\text{-}0 \vee \overline{\textbf{dead}\text{-}1}$) $\wedge$
23. ($\overline{\text{open-right-door-1}} \vee \overline{\textbf{dead}\text{-}0} \vee \textbf{dead}\text{-}1$) $\wedge$
24. ($\overline{\text{open-right-door-1}} \vee \textbf{dead}\text{-}0 \vee \overline{\textbf{tiger-behind-left-door}\text{-}0} \vee \overline{\textbf{dead}\text{-}1}$) $\wedge$
25. ($\overline{\text{open-right-door-1}} \vee \textbf{dead}\text{-}0 \vee \textbf{tiger-behind-left-door}\text{-}0 \vee \textbf{dead}\text{-}1$) $\wedge$
26. ($\overline{\text{listen-for-tiger-1}} \vee \overline{\textbf{rewarded}\text{-}0} \vee \textbf{rewarded}\text{-}1$) $\wedge$
27. ($\overline{\text{listen-for-tiger-1}} \vee \textbf{rewarded}\text{-}0 \vee \overline{\textbf{rewarded}\text{-}1}$) $\wedge$
28. ($\overline{\text{open-left-door-1}} \vee \overline{\textbf{tiger-behind-left-door}\text{-}0} \vee \overline{\textbf{rewarded}\text{-}1}$) $\wedge$
29. ($\overline{\text{open-left-door-1}} \vee \overline{\textbf{tiger-behind-left-door}\text{-}0} \vee \textbf{rewarded}\text{-}1$) $\wedge$
30. ($\overline{\text{open-right-door-1}} \vee \overline{\textbf{tiger-behind-left-door}\text{-}0} \vee \textbf{rewarded}\text{-}1$) $\wedge$
31. ($\overline{\text{open-left-door-1}} \vee \textbf{tiger-behind-left-door}\text{-}0 \vee \overline{\textbf{rewarded}\text{-}1}$)

Fig. 4. The SSAT formula for a 1-step TIGER plan constrains the variable assignments.

3. proposition p is `True` at time step t if it was `True` at time step $t-1$ and the action taken at time step t does not make it `False`, *or* the action at time step t makes p `True` ($1 \leqslant t \leqslant T$).

Each deterministic initial condition and goal condition in the problem generates a unit clause in the CNF formula. The initial conditions in our example generate the clauses ($\overline{\textbf{dead}\text{-}0}$) and ($\overline{\textbf{rewarded}\text{-}0}$) and the goal conditions generate the clauses ($\overline{\textbf{dead}\text{-}1}$) and (**rewarded**-1). The fact that the tiger is behind each door with equal probability is encoded by making the the variable **tiger-behind-left-door**-0 a chance variable with associated probability of 0.5. The number of clauses thus generated is bounded by $2P$.

The second condition, mutual exclusivity of actions for each time step, generates one clause with a special "exactly-one-of" operator that ensures that one and only one of the literals in the clause is `True`. When a literal in an exactly-one-of action clause is set to `True`, the solver immediately sets all other literals (actions) in that clause to `False`. This type of clause can be simulated by a small collection of standard clauses: one clause specifies that *some* action must be taken, and a quadratic (in the number of actions) number of clauses specify that for each possible pair of action variables, one action variable must be `False`. This approach, however, requires a more time-consuming series of unit propagations to set the other action literals in a clause to `False` when one of them is set to `True`. For this reason, we found that exactly-one-of clauses led to more compact and more efficiently solved encodings.

The third condition, effects of actions on propositions, generates one clause for each path through each decision tree in each action. Because of the transformation described at the beginning of Section 7.1.2, the probability of each leaf is either 0.0 or 1.0, and this path generates a single clause modeling the action's deterministic impact on the proposition given the circumstances described by that path. Note, however, that if these circumstances include a random proposition (described in Section 7.1.2), the net impact of the action modeled by the clause will be probabilistic. An example will clarify this process.

Fig. 2 shows the ordered list of decision trees associated with the listen-for-tiger action. The second decision tree describes the impact of the listen-for-tiger action on the **hear-tiger-behind-left-door** proposition. The left path of the tree specifies that when **tiger-behind-left-door** is `True`, the probability that **hear-tiger-behind-left-door** is `True` is 0.85. Since the probability in the leaf is strictly between 0.0 and 1.0, the converter generates a chance variable associated with this probability ($\mathbf{s}_1$). This path in the decision tree results in two clauses, one describing the impact of the action if the chance variable is `True` and one describing its impact if the chance variable is `False`. For the 1-step plan, this path generates the following two implications:

$$\text{listen-for-tiger-1} \wedge \textbf{tiger-behind-left-door}\text{-0} \wedge \mathbf{s}_1\text{-1} \rightarrow \textbf{hear-tiger-behind-left-door}\text{-1}$$

$$\text{listen-for-tiger-1} \wedge \textbf{tiger-behind-left-door}\text{-0} \wedge \overline{\mathbf{s}_1\text{-1}} \rightarrow \overline{\textbf{hear-tiger-behind-left-door}\text{-1}}$$

Note that a chance variable has the same time index as the action it modifies. Negating the antecedent and replacing the implication with a disjunction produces clauses 6 and 7 (Fig. 4):

$$\overline{\text{listen-for-tiger-1}} \vee \overline{\textbf{tiger-behind-left-door}\text{-0}} \vee \overline{\mathbf{s}_1\text{-1}} \vee \textbf{hear-tiger-behind-left-door}\text{-1}$$

$$\overline{\text{listen-for-tiger-1}} \vee \overline{\textbf{tiger-behind-left-door}\text{-0}} \vee \mathbf{s}_1\text{-1} \vee \overline{\textbf{hear-tiger-behind-left-door}\text{-1}}$$

Fig. 4 shows the complete formula for a 1-step plan.

The total number of action-effect clauses is bounded by $2T\sum_{i=1}^{A} L_i$ where L_i is the number of leaves in the decision trees of action i, so the total number of clauses C is bounded by

$$2P + T + 2T\sum_{i=1}^{A} L_i,$$

which is a low-order polynomial in the size of the problem. The average clause size is dominated by the average path length of all the decision trees.

Note that by using a compact representation of a factored state space, such as the ST representation, and translating that representation directly into SSAT form, we preserve the compactness of such a representation in our SSAT formula. The alternative—using a flat state space in which states are simply enumerated without regard to their characteristics, encoding states as propositions, and encoding in our clauses the impact of each action on each possible state—would be prohibitively expensive.

Also note that fixing a plan horizon does not prevent ZANDER from solving planning problems in which the horizon is unknown. By using *iterative lengthening*, a process in which successive instances of the planning problem with increasing horizons are solved, the optimal plan horizon can be discovered dynamically. We have not yet determined the feasibility of *incremental iterative lengthening*, a more sophisticated approach, in which the current instance of the planning problem with horizon T is incrementally extended to the instance with horizon $T + 1$ and earlier results are reused to help solve the extended problem.

7.1.4. Explanatory frame axioms

The example encoding in Fig. 4 uses *classical frame axioms*. If a state proposition is unaffected by an action, there are clauses that explicitly model this (e.g., clauses 12 through 17 model the fact that none of the actions can change **tiger-behind-left-door**). Since actions typically affect only a relatively small number of state propositions, thus generating a large number of classical frame axioms, we replace the classical frame axioms with *explanatory frame axioms*. Explanatory frame axioms generate fewer clauses by encoding possible *explanations* for changes in a proposition. For example, if the truth value of proposition p changes from `True` to `False`, it must be because some action capable of inducing that change was executed; otherwise, the proposition remains unchanged:

$$p^{t-1} \wedge \overline{p}^{t} \rightarrow a_1^t \vee a_3^t,$$

where a_1 and a_3 are the only actions that can cause the proposition p to change from `True` to `False`, and superscripts refer to time indices. We call these "simple" explanatory frame axioms because they do not make distinctions among the *possible* effects of an action. Unlike deterministic, unconditioned actions, it may be that, under certain circumstances, a_3 leaves p unchanged; its presence in the above list merely states that there is a set of circumstances under which a_3 *would* change p to $\overline{p}$. Thus, our simple explanatory frame axioms are similar to the frame axioms proposed by Schubert [67] for the situation calculus in deterministic worlds, and like his frame axioms, depend on the *explanation closure assumption*: that the actions specified in the domain specify all possible ways that

propositions can change. Details regarding this and other alternative SSAT encodings of probabilistic planning problems are available elsewhere [51].

Using explanatory frame axioms not only reduces the size of the encoding significantly in many cases, but sometimes produces clauses specifying invariants, which can speed up the SSAT solution process. The frame axioms for **tiger-behind-left-door** in Fig. 4 (clauses 12 through 17) illustrate both of these possibilities. Since there are *no* actions that can change the truth value of this state proposition, we can replace the classical frame axioms with the following explanatory frame axioms:

$$(\textbf{tiger-behind-left-door}\text{-}0 \vee \overline{\textbf{tiger-behind-left-door}\text{-}1}) \wedge$$
$$(\overline{\textbf{tiger-behind-left-door}\text{-}0} \vee \textbf{tiger-behind-left-door}\text{-}1)$$

which not only reduces the number of clauses from six to two, but makes explicit that the truth value of **tiger-behind-left-door** does not change.

We used simple explanatory frame axioms for all the SSAT encodings of the domains in Section 8.

7.2. Solving the SSAT encodings

The SSAT solution unit of ZANDER is a C++ program that takes as input an SSAT representation of a planning problem and finds an assignment *tree* that specifies the optimal choice-variable assignment given all possible settings of the observation variables. The assignment tree can be exponential in the size of the problem. The most basic variant of the solver follows the variable ordering exactly, constructing a binary tree of all possible assignments. Fig. 5 depicts such a tree; each node contains a variable under consideration, and each path through the tree describes a plan-execution history, an instantiation of the domain uncertainty, and a possible setting of the state variables. The tree shows the first seven variables in the ordering for the 2-step TIGER problem: the three choice variables encoding the action at time step 1— listen-for-tiger-1, open-left-door-1, open-right-door-1, the single observation chance variable **hear-tiger-behind-left-door**-1, and the three choice variables encoding the action at time step 2—listen-for-tiger-2, open-left-door-2, open-right-door-2. The root node of the tree contains the variable listen-for-tiger-1, the two nodes on the next level of the tree contain the variable open-left-door-1, and so forth (triangles indicate subtrees for which details are not shown). The observation variable **hear-tiger-behind-left-door**-1 is a branch point; the optimal assignment to the remaining choice variables (listen-for-tiger-2, open-left-door-2, open-right-door-2) will be different for different values of this variable.

This representation of the planning problem is similar to *AND/OR trees* and *MINIMAX trees* [55]. Choice variable nodes are analogous to OR, or MAX, nodes, and chance variable nodes are analogous to AND, or MIN, nodes. However, the probabilities associated with chance variables (our opponent is nature) make the analogy somewhat inexact. Our trees are more similar to *MINIMAX trees with chance nodes* [1] but without the MIN nodes—instead of a sequence of alternating moves by opposing players mediated by random events, our trees represent a sequence of moves by a single player mediated by the randomness in the planning domain.

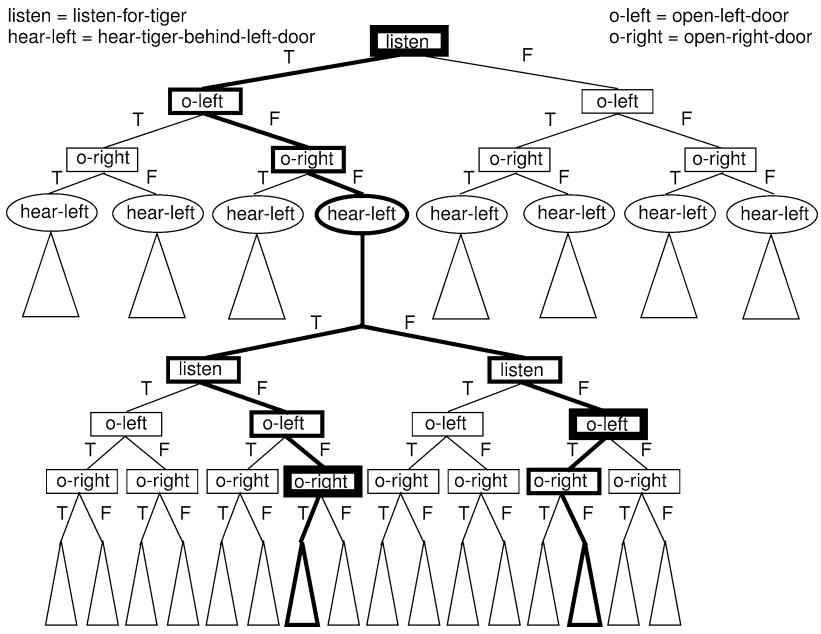


Fig. 5. ZANDER selects an optimal subtree.

The solver essentially implements the DPLL-based algorithm described in Section 5.1. It does a depth-first search of the tree, constructing a solution subtree by calculating, for each node, the probability of a satisfying assignment given the partial assignment so far. For a choice variable, this is a maximum probability and produces no branch in the solution subtree; the solver notes which value of the variable yields this maximum. For a chance variable, the probability will be the probability weighted average of the satisfaction probabilities for that node's subtrees. For an observation variable, the probability will be the sum of the satisfaction probabilities for that node's subtrees and will produce a branch point in the solution subtree. The solver finds the optimal plan by determining the subtree with the highest probability of success. In Fig. 5, the plan portion of this subtree appears in bold, with action choices (action variables set to `True`) in extra bold. The optimal plan is: listen-for-tiger; if **hear-tiger-behind-left-door** is `True`, open-right-door; if `False`, open-left-door.

In contrast to heuristic search approaches, which must follow a prescribed variable ordering, ZANDER can consider variables out of the quantifier ordering specified in the SSAT problem when this allows it to prune subtrees. The main novelty of our approach, in fact, lies in our use of the stochastic satisfiability formulation of the problem, which allows ZANDER to use satisfiability heuristics, such as unit propagation and pure variable elimination, to prune subtrees. It is possible that the algorithm could use heuristic search to solve the trees generated by our planning problems. A worthwhile area of research would be to compare the performance of these two approaches and attempt to develop techniques that combine the advantages of both.

8. Results

This section reports results comparing ZANDER to seven other planning techniques. All experiments were conducted on an 866 MHz Dell Precision 620 with 256 Mbytes of RAM, running Linux 7.2. We note here that the timings reported below do not include the time needed for any of the planners to translate the initial problem representation into the form necessary for the planner's solver. This means, for example, that the $\mathcal{PPL}$-to-SSAT translation times for ZANDER are not included, the parse and compile times required for GPT to create the C program that solves the planning problem are not included, and the time to translate a user-friendly POMDP specification into the required form for POMDP:INC_PRUNE is not included. These costs tend to be small and generally do not grow rapidly with horizon length.

8.1. Sample domains

The problems selected cover a range of different possibilities with respect to initial conditions (deterministic, nondeterministic, or probabilistic) and the effects of causal actions (deterministic, nonprobabilistic, or probabilistic). Observability is complete or (usually) partial. In all cases when the observability is partial, the agent can never observe the state completely in a single step.

In the GO-5 domain (General Operations, 5 operations), adapted from a problem described by Onder [56], there are deterministic initial conditions, probabilistic causal actions, and partial, but completely accurate, observability. There are five actions—paint, clean, polish, trim, and vacuum—each of which produces a single desired effect with probability 0.5. Initially none of the effects have been achieved; the goal conditions require that all these effects be accomplished without falling into an error condition, which results when the agent attempts to execute an action whose effect has already been achieved. The agent is able to observe the effect of any action just executed with complete accuracy.

In the MEDICAL-5ILL domain [72], there are probabilistic initial conditions, deterministic causal actions, and partial, but completely accurate, observability. A patient is either healthy or has one of five illnesses (with equal probability). Fortunately, there is a medication for each illness that will cure the patient with certainty. The patient, however, will die if she receives any medication for which she does not have the corresponding illness. Thus, it is critical to disambiguate the initial conditions. There is a stain test that allows the agent to determine which of the following three categories the patient's illness falls into: (1) illness 1 or illness 2, (2) illness 3 or illness 4, or (3) illness 5. There is a white cell count test that allows the agent to distinguish between illnesses 1 and 2 and illnesses 3 and 4. Together, these tests allow the agent to determine the patient's illness with certainty and administer the correct medication.

The COFFEE-ROBOT domain, a slightly modified version of a domain described by Boutilier and Poole [12], contains nondeterministic or probabilistic initial conditions (depending on the planner's capabilities), deterministic causal actions, and partial, but completely accurate, observability. A robot must determine whether its user wants coffee and, if so, go to the cafeteria, buy some coffee, and return to the office and deliver it to the user. In addition, it might be raining and, since the robot should not get wet, it must take

Table 1
The size of the state space and the degree of observability varied among the tested domains

Domain	Number of:		
	Actions	State variables	Observation variables
GO-5	6	6	5
MEDICAL-5ILL	7	10	4
COFFEE-ROBOT (partially observable)	6	8	2
COFFEE-ROBOT (completely observable)	4	8	8
SHIP-REJECT	4	5	1
TIGER	3	3	1

an umbrella if it is raining.[6] If the user does not want coffee, the robot should do nothing, since having coffee when it is not wanted makes the user unhappy. Initially, it is uncertain whether it is raining and whether the user wants coffee. In versions of the problem that use probabilistic initial conditions, both these conditions are true with probability 0.50. The robot can ask the user if she wants coffee and can look out the window to see if it is raining. These observations are always accurate. All the other actions—get-umbrella, change-location, buy-coffee, and deliver-coffee—have probabilistic effects. Note that in order to make a better comparison between ZANDER and SPUDD (which assumes complete observability), we also created a completely observable version of this problem.

The SHIP-REJECT domain [19] has probabilistic initial conditions, probabilistic actions, and partial, noisy observability. A part is initially flawed (not visible) and blemished (visible, and perfectly correlated with flawed) with probability 0.30. The objective is to paint and process the part, where processing consists of deciding whether to ship the part (if not flawed) or reject the part (if flawed). While painting the part erases the blemish, it does not correct the internal flaw, so the agent must observe whether the part is blemished, paint the part, and then condition the ship/reject decision on its earlier observation.

The TIGER domain [30] has probabilistic initial conditions, deterministic actions, and partial, noisy observability. The agent is faced with two doors, one concealing a hungry tiger, the other concealing a treasure. The objective is to get the treasure. Before opening one of the doors, the agent can listen for the tiger, but this observation is only accurate with probability 0.85. A unique feature of this problem is that, in general, the agent needs to condition its actions on the entire observation history in order to act correctly.

Some idea of the size of these domains can be obtained from the Table 1, listing the number of actions, state variables, and observation variables in each domain.

[6] In the original version of this problem, the robot incurs a small cost for getting wet. In our version, the robot is required to stay dry. We could emulate a slight cost by introducing a small probability of failure for getting wet. Thus, the probability of goal achievement for a plan in which the robot gets wet will be non-zero, but less than that of a plan that keeps the robot dry.

Note that these numbers vary slightly among the planners due to minor changes made to accommodate the abilities of the planners. For example, ZANDER needs a noop action in the GO-5 domain to execute if the goal is achieved before the end of the fixed-length plan is reached, while the other planners that can handle this problem do not need such an action.

8.2. Planners

All of the planners we tested are able to solve planning problems with some degree of uncertainty. There is considerable variation, however, in the kinds of uncertainty they are capable of dealing with. The planner may only be able to cope with deterministic initial conditions (only one possible initial state), or it may be able to deal with nondeterministic initial conditions (a set of possible initial states) or probabilistic initial conditions (a probability distribution over possible initial states). The planner may be able to cope with partial observability (partial in scope and/or accuracy) or may assume complete observability. Finally, the planner may be able to reason about non-probabilistic actions (a list of possible outcomes) or probabilistic actions (a probability distribution over possible outcomes. Before briefly describing the seven planners we tested in addition to ZANDER, we summarize these characteristics in the following table. The version of ZANDER we used ran the SSAT solver described in Section 5.1 with $\theta_l = 0$ and $\theta_h = 1$.

The eight planners can be placed into three broad categories of approaches to probabilistic planning: the MDP/POMDP approach (GPT, POMDP:INC_PRUNE, HANSEN-FENG, and SPUDD), the classical causal-reasoning approach (MAHINUR), and the constraint-based approach (ZANDER, SGP, and PGRAPHPLAN).

GPT (General Planning Tool) is an integrated software package for modeling, analyzing, and solving planning problems that involve uncertainty and partial information [8]. It uses optimal heuristic search for conformant planning and real-time dynamic program-

Table 2
The eight tested planners vary in their abilities to handle different types of uncertainty

Planner	Type of initial conditions	Type of observability	Type of actions
ZANDER	Probabilistic	Partial	Probabilistic
GPT	Probabilistic	Partial	Probabilistic
POMDP:INC_PRUNE	Probabilistic	Partial	Probabilistic
HANSEN-FENG	Probabilistic	Partial	Probabilistic
SPUDD	Deterministic[a]	Complete	Probabilistic
MAHINUR	Probabilistic	Limited Partial[b]	Probabilistic
SGP	Nondeterministic	Limited Partial[c]	Non-probabilistic
PGRAPHPLAN	Deterministic[a]	Complete	Probabilistic

[a] Probabilistic initial conditions can be simulated by forcing an initial action that probabilistically sets the initial conditions.

[b] MAHINUR cannot currently handle multiple observations or a series of instances of the same observation although, in principle, it could do so.

[c] SGP cannot currently handle noisy observations.

ming (RTDP) [2] for nondeterministic (non-probabilistic), probabilistic, and contingent planning. RTDP is a version of dynamic programming that finds a policy by running a number of trials, each one starting in the initial state and ending either in a goal state or after some step limit has been reached. During a trial, the current approximation of the policy is used as a heuristic function to determine the action to be taken in a particular state, and that approximation is updated after every action. GPT attempts to construct a plan graph for the specified initial state that will reach a goal state with certainty.

POMDP:INC_PRUNE [14] uses a method of dynamic-programming updates called *incremental pruning* to solve POMDPS more efficiently. Value functions are represented as sets of vectors and it is crucial in a step of value iteration to be able to reduce such a set of vectors to its minimum size form. Incremental pruning sequences the vector purging operations involved in this process so as to reduce the number of linear programs that have to be solved and to reduce the number of constraints in the linear programs themselves. We ran POMDP:INC_PRUNE on the corresponding finite-horizon POMDP formulations of our domains; POMDP:INC_PRUNE attempts to produce a planning graph that specifies the course of action from any initial state that maximizes the expected reward.

The HANSEN-FENG algorithm [27] exploits a factored state representation to accelerate the incremental pruning algorithm for solving POMDPS. Based on a framework described by Boutilier and Poole [12], it uses *algebraic decision diagrams* (ADDs) to compactly represent the transition probabilities, value function, and reward function of a POMDP (ADDs are a generalization of binary decision diagrams that can be used to represent real-valued functions). This allows the pruning step involved in the dynamic programming solution of a POMDP to be implemented much more efficiently. We ran HANSEN-FENG on the corresponding finite-horizon POMDP formulations of our domains. Like POMDP:INC_PRUNE, HANSEN-FENG attempts to produce a planning graph that specifies the course of action from any initial state that maximizes the expected reward.

Note that we used a discount factor of 0.9 for both POMDP:INC_PRUNE and HANSEN-FENG. Although a discount factor of 1.0 would have been more appropriate for a comparison with ZANDER, these planners did not converge in a competitive amount of time using a discount factor of 1.0.

SPUDD [28] is a dynamic abstraction method for solving MDPS (and, thus, assumes complete observability). SPUDD uses ADDs to represent value functions and policies in a compact way. This compact representation allows SPUDD to perform value iteration efficiently enough to solve MDPS with tens of millions of states exactly. SPUDD attempts to produce an ADD that prescribes the best action to take in any given state. Repeated application of this ADD provides a plan to reach a goal state from any initial state.

MAHINUR [58,59], a contingent, probabilistic, partial-order planner combines BURIDAN's probabilistic action representation [41] and a system for managing these actions with a CNLP-style approach to handling contingencies. The novel feature of MAHINUR is that it identifies those contingencies whose failure would have the greatest negative impact on the plan's success and focuses its planning efforts on generating plan branches to deal with those contingencies. This selectivity in adding branches to the plan can boost MAHINUR's speed considerably (see MAHINUR's performance on the GO-5 domain), but Onder and Pollack [58] identify several domain assumptions (including a type of subgoal decomposability) that underlie the design of MAHINUR, and there are no guarantees on the

correctness of MAHINUR for domains in which these assumptions are violated. None of the problems in our test suite violate any of these assumptions, but we were unable to test MAHINUR on all the problems. Although MAHINUR provides a framework to reason about the relationship between observation actions (either the same observation action repeated or a sequence of different observation actions), this capability has not been implemented yet [57]. MAHINUR produces a contingent plan that reaches a goal state from a specified start state with a probability that meets or exceeds a specified threshold.

SENSORY GRAPHPLAN (SGP) [72] is based on GRAPHPLAN [5] (see Section 2.1). SGP deals with uncertainty by constructing a planning graph that captures all possible worlds the agent could be in at any given time. SGP constructs plans with sensing actions that gather information to be used later in distinguishing between different plan branches. However, SGP has not been extended to handle probabilistic actions and noisy observations, so it is only applicable to two of the domains tested (MEDICAL-5ILL and COFFEE-ROBOT). SGP produces a contingent plan that reaches a goal state from a specified start state with certainty.

PGRAPHPLAN [7], also based on GRAPHPLAN, employs forward search through the planning graph to find a contingent plan with the highest expected utility. PGRAPHPLAN operates in the MDP framework (complete observability). PGRAPHPLAN does forward dynamic programming using the planning graph as an aid in pruning search. We note here that ZANDER essentially does the same thing by following the action/observation variable ordering specified in the SSAT problem. When ZANDER instantiates an action, the resulting simplified formula implicitly describes the possible states that the agent could reach after this action has been executed. If the action is probabilistic, the resulting subformula (and the chance variables in that subformula) encodes a probability distribution over the possible states that could result from taking that action. And the algorithm is called recursively to generate a new implicit probability distribution every time an action is instantiated. PGRAPHPLAN returns a contingent plan to reach a specified set of goal states from a specified initial state (if such a plan exists).

8.3. Comparisons between planning techniques

Many factors make it difficult to do a straightforward comparison of these eight planners, and a good deal of caution must be exercised in interpreting the results of our experiments.

- As described above, not all of the planners are attacking the same type of planning problems (e.g., in degree of observability).
- The planners are developed to varying degrees of their potential (e.g., MAHINUR cannot currently handle multiple observations although, in principle, it could).
- They use different state representations (a flat representation for POMDP:INC_PRUNE, factored for all the others), and their problem representation languages allow different types of problem information to be expressed (e.g., ZANDER allows irreversible conditions to be stated explicitly).
- The planners produce different kinds of output. ZANDER, MAHINUR, and PGRAPHPLAN produce a contingent plan that will reach a goal state from the specified initial

conditions with highest probability. GPT produces a controller for the specified initial conditions that will succeed with certainty. POMDP:INC_PRUNE, HANSEN-FENG, and SPUDD produce a universal controller that maximizes expected reward. SGP finds a contingent plan that will succeed with certainty. Due to this variability in output, we have not included an exhaustive comparison of the quality of the plans produced, describing rather instances where differences in plan quality seemed noteworthy.

- The planners are not all written in the same language and may have been optimized to differing degrees. Most of them are written in C and C++; MAHINUR and SGP are written in LISP.

In addition, although we tried to use each planner to its best advantage, we were probably unsuccessful due to limited familiarity with the planners (except ZANDER). In several cases, the developers of the algorithm pointed out better problem formulations or provided a working formulation where we had been unable to construct one. For all these reasons, it is probably most useful to view these experiments as an exploration of a number of planners currently being developed that can deal with uncertainty. Note that all times are in CPU seconds.

Fig. 6 plots running time versus horizon length (number of steps in the plan) for four of the test domains and all applicable planners. Fig. 6(a) shows the results for GO-5. GPT and SPUDD produced plans that are not dependent on horizon length, and so are shown as straight lines. SPUDD ran in under a second, and GPT took over a half an hour because of memory limitations. ZANDER and POMDP:INC_PRUNE (discount factor 0.9) had running times that grew dramatically with horizon, although ZANDER demonstrated faster running times and better scaling properties on this problem. PGRAPHPLAN's time to plan grew so slowly with horizon that at a horizon length of 1000 (not shown) it took about 4 seconds. MAHINUR carried out a series of plan refinements (not actually plan extensions, as suggested by the graph). In fact, in a direct comparison of time versus plan-success probability, MAHINUR dominated ZANDER on this problem by about an order of magnitude. For obscure reasons, HANSEN-FENG did not run properly on our encoding of this problem.

Of the planners applicable to MEDICAL-5ILL (Fig. 6(b)), SGP and GPT do not depend on horizon length; GPT ran in about a tenth of a second here, and SGP took about a minute and a half. ZANDER and POMDP:INC_PRUNE (discount factor 0.9) had running times that again grew dramatically with horizon. Note, however, that a 3-step plan is sufficient to guarantee goal attainment on this problem and, at this horizon, ZANDER finds the optimal plan more quickly than the other planners (0.01 s). Again, ZANDER ran more quickly than POMDP:INC_PRUNE, and POMDP:INC_PRUNE experienced a segmentation fault on the 10-step plan. Once again, for obscure reasons, HANSEN-FENG did not run properly on this problem. We think this is a problem with our use of the representation used by HANSEN-FENG, not a failure of the algorithm.

The COFFEE-ROBOT problem (Fig. 6(c)) was the most general and largest problem we tested. POMDP:INC_PRUNE (discount factor 0.9) and ZANDER were able to solve the problem; HANSEN-FENG (discount factor 0.9) produced an incorrect controller, due to either a modeling error we could not track down or a problem with the planner itself. ZANDER ran significantly faster than POMDP:INC_PRUNE (460 s v. 3600 s for an 8-

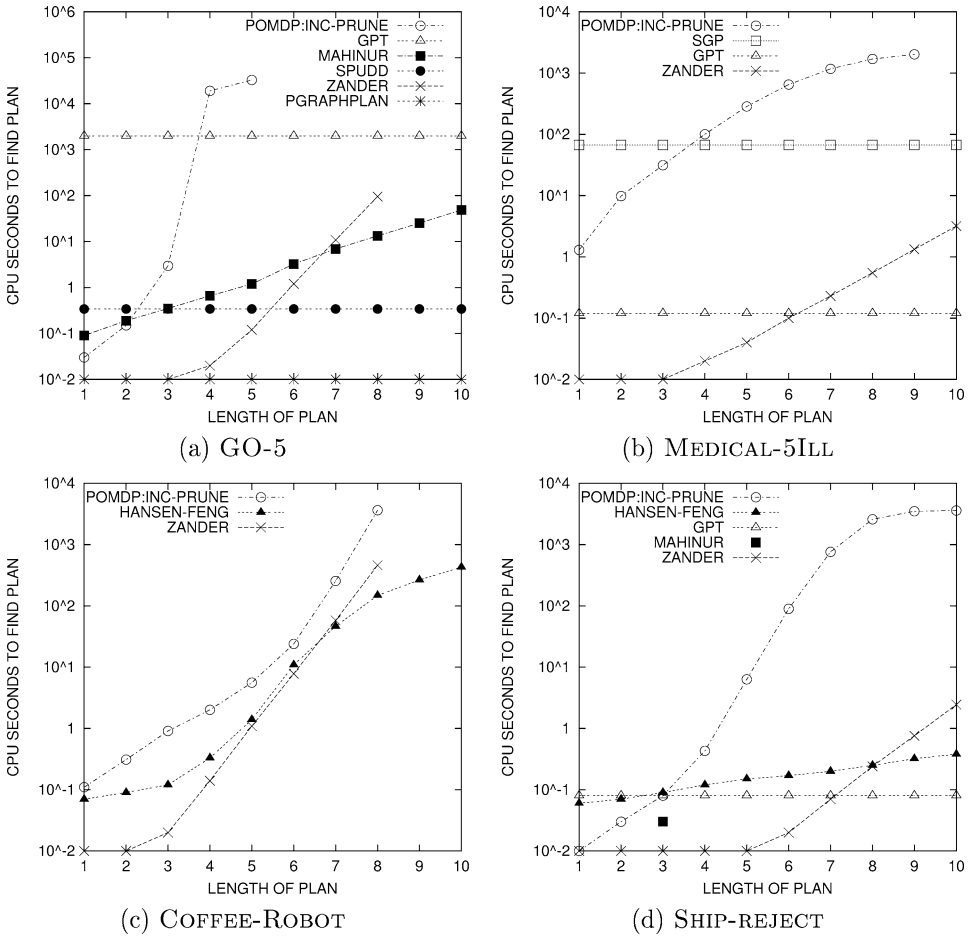


Fig. 6. Each applicable planner was run on each of the test domains.

step plan). While both MAHINUR and GPT were applicable to this problem, MAHINUR crashed for unknown reasons and GPT ran out of memory while planning. SPUDD solved a completely observable version of the problem in 0.60 s; adding observability reduced ZANDER's solution time by 56% for an 8-step plan.

On the SHIP-REJECT problem (Fig. 6(d)), MAHINUR produced a 3-step plan in 0.03 s, but could not produce a better, longer plan due to the implementation limitations cited in Section 8.2. GPT's plan was produced in 0.08 s, but this plan succeeds only with probability 0.9975. (ZANDER needed 0.02 s to produce a comparable plan and was able to produce plans that succeeded with a higher probability.) We attempted to coax GPT to produce a plan that succeeded with higher probability by increasing the cost of failure, but this succeeded only in increasing the solution time in direct proportion to the cost. POMDP:INC_PRUNE scaled badly with horizon, but tapered off after about 8 steps. ZANDER took about a hundredth of a second to produce plans up to 5 steps, then ran into

trouble with longer plans. HANSEN-FENG took longer at first, but appeared to scale well. The plans produced by HANSEN-FENG, however, never paint the part more than once, thus capping the probability of success at 0.9215, whereas ZANDER produces plans that succeed with a higher probability as the horizon increases.

We also ran comparisons on TIGER (not shown), which followed a very similar pattern. GPT took about a tenth of a second. ZANDER took about a hundredth of a second for plan lengths up to 5. Both ZANDER (at horizons greater than 5) and POMDP:INC_PRUNE scaled poorly. HANSEN-FENG, although starting out slower than ZANDER and POMDP:INC_PRUNE, scaled better (but always produced a segmentation fault just after finding the plan).

In principle, there are efficient conversion procedures to translate planning domains from one representation to equivalent planning domains in another. However, we frequently found that the resulting domains violated assumptions built into the design of the planners, rendering these formulations unworkable. When this happened, we tweaked the representations by hand. However, in the TIGER domain, these tweaks significantly changed the domain, causing the resulting plans to differ from planner to planner. Of course, this undercuts the usefulness of the empirical comparison, as the planners were solving different problems.

8.4. General observations

Although our test domains varied quite a bit, the results we saw were fairly consistent, ignoring differences in implementation and output. When applicable, PGRAPHPLAN was the fastest, with SPUDD, MAHINUR, GPT, HANSEN-FENG, SGP, and POMDP:INC_PRUNE behind it, in order. The relative performance of ZANDER varied with horizon. At a horizon length of about 5, ZANDER was the second fastest, whereas at a horizon length of about 10, it fell to sixth fastest.

We believe all the planners have strengths and limitations. For example, the two planners that turn in exceptionally good performances on some problems—SPUDD and PGRAPHPLAN—are the two planners that assume complete observability. GPT runs best if the number of reachable information states is finite and relatively small, while POMDP:INC_PRUNE runs into trouble if the number of undominated plans grows exponentially.

ZANDER also appears better suited to some problems than others. It appears to work best when:

1. not many plans have the same probability of success (leads to more effective pruning, unlike GO-5),
2. each proposition is changed by relatively few actions (leads to few clauses),
3. paths in decision trees are not too long (leads to shorter clauses, providing more opportunities for ZANDER's SSAT heuristics to operate), and
4. few action effects are probabilistic (leads to fewer branches to consider in the search tree, unlike COFFEE-ROBOT).

Our ongoing exploration will likely bring additional insight into expanding the applicability of ZANDER and increasing our understanding of its appropriateness for specific domains.

9. Further work

ZANDER, like most of the planners we tested, exploits the state information available in a factored state space to efficiently solve planning problems in stochastic, partially observable domains. Because ZANDER can encode any degree of observability (both in terms of which state propositions can be observed, and how accurately they can be observed) and because ZANDER does not limit the size (only the horizon) of the resulting plan, ZANDER can solve arbitrary, goal-oriented, finite-horizon, factored POMDPS. This is in sharp contrast to SPUDD, MAHINUR, SGP, and PGRAPHPLAN, all of which are limited in the types of problems they can handle.

Although an exact assessment is impossible due to the differences among planners discussed in Section 8.3, ZANDER appears competitive with all the planners we tested. These results are especially encouraging, given that there are a number of improvements to ZANDER that have shown promise for scaling up to larger problems.

9.1. Improvements to ZANDER

Given ZANDER's two-phase approach, these improvement naturally fall into two categories:

- improvements in the SSAT encoding of planning problems, and
- improvements in the algorithm for solving the SSAT encodings.

In the following sections, we will describe each of these improvements and discuss initial efforts to implement them in ZANDER.

9.1.1. Improved SSAT encodings

The encodings with explanatory frame axioms used in this paper were developed by Majercik and Rusczek [51]; several other alternative SSAT encodings, including parallel-action encodings, were also described in that paper. But, even more efficient SSAT encodings like those with explanatory frame axioms suffer from the fact that they frequently contain clauses at a particular time step that are superfluous since they describe the effects of an action that cannot be taken at that time step (or will have no impact if executed). The first author is currently working on an approach that is analogous to the GRAPHPLAN [6] approach of incrementally extending the depth of the planning graph in the search for a successful plan. The idea is to build the SSAT encoding incrementally, attempting to find a satisfactory plan in t time steps (starting with $t = 1$) and, if unsuccessful, using the knowledge of what state the agent *could* be in after time t to guide the construction of the SSAT encoding for the next time step. This reachability analysis would not only prevent superfluous clauses from being generated, but would also

make it unnecessary to pick a plan length for the encoding, and would give the planner an *anytime* capability, producing a plan that succeeds with *some* probability as soon as possible and increasing the plan's probability of success as time permits.

Kautz et al. [31] note that it is possible to use resolution to eliminate any subset of variables in a SAT formula, but that this usually leads to an exponential blowup in the number of clauses in the encoding. For GRAPHPLAN-based encodings, however, eliminating the propositional variables that describe the state of the environment leads to an increase that is polynomial in the number of these propositions. Although we have not conducted extensive tests, our SSAT solver seems to be more sensitive to the number of variables than to the number of clauses. It is possible that the efficiency of the solver could be improved as the result of identifying a group of variables whose elimination would entail only a polynomial increase in the number of clauses.

Domain-specific knowledge could be exploited in either the construction of the SSAT formula or its subsequent solution. The first approach has been explored by Kautz and Selman [36] in the context of SATPLAN. In their work, four types of clauses that can be added to a SAT encoding of a planning problem were described:

- *Conflict* clauses and *derived effect* clauses implied by the domain's action descriptions.
- *State invariant* clauses implied by the domain's actions and initial conditions.
- *Optimality condition* clauses implied by the actions, initial conditions, and plan length.
- *Simplifying assumption* clauses.

The first three types of clauses make knowledge that was previously implicit in the problem domain explicit and are analogous to providing lemmas to a theorem prover. The fourth type of clause is not implicit in the domain and, in fact, may prevent some solutions from being found [36]. Adding such clauses to the SAT encoding can accelerate the solution process enormously, particularly for systematic satisfiability testers, reducing the solution time on some problems from in excess of 48 hours to a few seconds [36].

Another way of incorporating domain-specific knowledge is to use such knowledge to guide the SSAT solution process. For example, we might be able to use optimality criteria or means-ends analysis to efficiently identify high probability plans or prune low probability plans.

9.1.2. Improved SSAT solution techniques

More sophisticated data structures in which to store the CNF encoding would almost certainly improve the efficiency of the solver. For example, the *trie* data structure has been used to represent SAT problems, and several advantages have been claimed for this approach [73], including automatic elimination of duplicate clauses when the trie is constructed, reduced memory requirements, and more efficient unit propagation.

The current splitting heuristic orders groups of candidate variables according to the order of their appearance in the quantifier ordering. In the plan-execution history segment (variables encoding actions and observations), this coincides with the ordering that would be imposed by time-ordered splitting (give priority to variables with lower time indices). The chance variables in the domain-uncertainty segment and the choice variables in the

segment that encodes the result of the plan-execution history given the domain uncertainty are time-ordered.

The current heuristic, however, does not specify an ordering for variables within the blocks of similarly quantified variables that have the same time index. This may be insignificant in small problems, but in real-world problems with a large number of variables at each time step, a splitting heuristic that addresses this subordering issue could provide a significant performance gain.

ZANDER separately explores and saves two plan execution histories that diverge and remerge, constructing a plan tree when a directed acyclic graph would be more efficient. ZANDER should be able to memoize subplan results so that when it encounters previously solved subproblems, it can merge the current plan execution history with the old history. Memoization boosted MAXPLAN's performance tremendously [50] and it is likely that it would have a similar beneficial effect on ZANDER's performance.

ZANDER could probably be improved by adapting other techniques that have been developed for constraint satisfaction problems (CSPs). In CSP terms, ZANDER uses backtrack search with forward checking and a variable ordering heuristic that gives priority to unit-domained variables. We would like to explore the possibility of incorporating CSP look-back techniques, such as backjumping and learning (deriving no-goods) [3]. Perhaps a more direct way of exploiting the connection to CSPs is to model planning problems using stochastic constraint satisfaction [70], as this provides a more direct way of expressing multivalued domain variables.

9.2. Extending ZANDER

The improvements discussed in the sections above focus on accelerating ZANDER's performance. An extension to ZANDER that would significantly broaden the scope of planning problems it is able to handle is the ability to produce more complex plans.

ZANDER produces acyclic, contingent plans. This is a significant improvement over straight-line plans, but it is not hard to think of planning domains in which the only realistic plan is a looping plan, in which an action—or sequence of actions—is repeated an indefinite number of times until some effect is achieved. We would like to extend ZANDER to be able to produce looping plans. The problem of finding such plans is still in PSPACE [44], so it is possible that ZANDER could be extended to find such plans.

One possibility is suggested by C-MAXPLAN, a less successful contingent planning extension of MAXPLAN [47]. In one version of C-MAXPLAN, instead of searching for the optimal contingent plan of a given length, the algorithm searches for an optimal small *policy* to be applied for a given number of steps. Perhaps the SSAT encodings of ZANDER could be modified to generate policy-like solutions as well. Such solutions would allow ZANDER to specify plans in which an action is to be repeated as many times as is necessary, up to the step limit specified. If no successful policy could be found for a given step limit, because a particular action could not be repeated often enough, iteratively increasing the step limit would eventually lead to a successful combination of policy and step limit.

9.3. Approximation techniques for solving SSAT problems

Although improvements to the current planner may allow ZANDER to scale up to problems of moderate complexity, they are unlikely to be sufficient to achieve our ultimate goal of planning efficiently in large, real-world domains. We think it is likely that we will need to develop an approximation technique for solving SSAT problems to scale up to problems of this size. Optimality is sacrificed for "anytime" planning and performance bounds, and although this may not improve worst-case complexity, it is likely to help for typical problems.

The first author is currently developing APROPOS^2, a probabilistic contingent planner based on ZANDER that produces an approximate contingent plan and improves that plan as time permits [48]. APROPOS^2 does this by considering the most probable situations facing the agent and constructing a plan, if possible, that succeeds under those circumstances. Given more time, less likely situations are considered and the plan is revised if necessary. In some cases, a plan constructed to address a relatively low percentage of possible situations will succeed for situations not explicitly considered as well, and may return an optimal or near-optimal plan. This means that APROPOS^2 can sometimes find optimal plans faster than ZANDER. And the anytime quality of APROPOS^2 means that suboptimal plans could be efficiently derived in larger time-critical domains where ZANDER might not have time to calculate the optimal plan.

Another possibility is to convert the probabilistic planning problem into a deterministic planning problem by rounding each decision-tree leaf probability to 0.0 or 1.0, solving the resulting deterministic planning problem relatively efficiently and then gradually reintroducing uncertainty to improve the quality of the solution. It is not clear, however, how to reintroduce the uncertainty without sacrificing the efficiency gained by removing it.

ZANDER systematically searches for satisfying assignments by setting the truth value of each variable in turn and considering the remaining subformula. This is significantly different from the WALKSAT approach in SATPLAN, which begins with a complete truth assignment and adjusts it through stochastic local search to achieve a satisfying assignment. In the same way that stochastic local search can solve much larger SAT problems than systematic search (in general), it is possible that adapting stochastic local search to the solution of SSAT problems would provide significant performance gains. The fact that an SSAT solver needs to systematically evaluate all possible assignments to solve the SSAT problem exactly argues for a systematic approach. There are, however, a number of ways that stochastic local search could be incorporated into an SSAT solver [52].

One possible use for an approximation technique is in a framework that interleaves planning and execution, in order to scale up to even larger domains than approximation alone could attack. The idea here would be to use the approximation technique to calculate a "pretty good" first action (or action sequence), execute that action or action sequence, and then continue this planning/execution cycle from the new initial state (see, for example, the work of [38]). This approach could improve efficiency greatly (at the expense of optimality) by focusing the planner's efforts only on those contingencies that actually materialize.

10. Summary

Probabilistic planning attempts to merge traditional artificial intelligence planning (propositional representations of large domains) with operations research planning (stochastic modeling of uncertainty) to produce systems that can reason efficiently about plans in complex, uncertain applications. Our approach to probabilistic planning is rooted in the planning-as-satisfiability paradigm, in which the specification of a planning problem is "compiled" to its computational core in the form of an equivalent Boolean satisfiability problem.

Our planner, ZANDER, accepts propositional representations of partially observable Markov decision processes, making it highly general. It can cope with initial-state uncertainty, observation uncertainty, and transition uncertainty. Although it is tuned to solve goal-oriented problems, it can be used for more general reward-maximizing applications as well. ZANDER directly converts planning problems into a stochastic satisfiability format, which can be solved relatively quickly using a general purpose stochastic satisfiability solver. Due to the generality of the satisfiability representation, it would be easy to extend ZANDER to model, for example, extrinsic events and factored actions by changing the conversion module only. Although ZANDER is still far from solving a wide range of practical problems, it represents a promising new direction in domain independent planning under uncertainty.

Acknowledgements

We would like to thank Blai Bonet, Zhengzhu Feng, Eric Hansen, Henry Kautz, Donald Loveland, Nilufer Onder, Mark Peot, Toni Pitassi, and our anonymous reviewers for their help during this research and the preparation of this article.

This work was funded in part by NASA Ames Research Center through a Graduate Student Researchers Program Fellowship to Stephen Majercik, and by the National Science Foundation through a Career Grant to Michael Littman (NSF grant IRI-9702576).

Appendix A

Theorem 1. *Any probabilistic planning problem with a discounted expected reward criterion can be reformulated in polynomial time using a probability of goal achievement criterion.*

Proof. Consider a probabilistic problem with a discounted expected reward criterion defined by a set of states S, a set of actions A, transition probabilities $T(s, a, s')$, rewards $R(s, a)$, and discount factor $\gamma < 1$. Assume without loss of generality all rewards are in the range $0 \leqslant R(s, a) < 1-\gamma$. (Any affine transformation of the rewards leads to an equivalent probabilistic planning problem with identical optimal and approximately optimal policies.)

We now define an equivalent probabilistic planning problem with a probability of goal achievement criterion. The new problem uses the same action space $A^* = A$ and a state

space $S^* = S \cup \{\text{goal}, \text{sink}\}$, where the sink state cannot be escaped and the goal state is the goal. Define the transitions

$$\begin{aligned} T^*(s,a,\text{goal}) &= R(s,a), \\ T^*(s,a,\text{sink}) &= (1-\gamma) - R(s,a), \\ T^*(s,a,s') &= \gamma T(s,a,s'), \quad \text{for all } s \text{ and } s' \text{ in } S, a \text{ in } A. \end{aligned}$$

According to this definition, on each step the system terminates with probability $1-\gamma$ and continues with probability γ. When it terminates, the goal probability is proportional to $R(s,a)$. Under most reasonable representation schemes, including dynamic Bayes' nets and the sequential-effects-tree representation, a representation of T^* can be created from the representation of T in time linear in its size.

The value function for a policy π in the original probabilistic planning problem with a discounted expected reward criterion is the unique solution to the system of equations:

$$V_\pi(s) = R\big(s,\pi(s)\big) + \sum_{s' \in S} \gamma T\big(s,\pi(s),s'\big) V_\pi(s'). \tag{A.1}$$

The value function for a policy π in the revised probabilistic planning problem with a probability of goal achievement criterion is the unique solution to the system of equations:

$$\begin{aligned} V^*_\pi(s) &= T^*\big(s,\pi(s),\text{goal}\big) + \sum_{s' \in S} T^*\big(s,\pi(s),s'\big) V^*_\pi(s') \\ &= R\big(s,\pi(s)\big) + \sum_{s' \in S} \gamma T\big(s,\pi(s),s'\big) V^*_\pi(s'). \end{aligned} \tag{A.2}$$

Note that Eqs. (A.1) and (A.2) define the same value function, showing that the value of a policy in the revised probabilistic planning problem is precisely the same as that of the original problem; they are equivalent.

References

[1] B.W. Ballard, The *-minimax search procedure for trees containing chance nodes, Artificial Intelligence 21 (3) (1983) 327–350.

[2] A.G. Barto, S.J. Bradtke, S.P. Singh, Learning to act using real-time dynamic programming, Artificial Intelligence 72 (1) (1995) 81–138.

[3] R.J. Bayardo Jr, R.C. Schrag, Using CSP look-back techniques to solve real-world SAT instances, in: Proceedings of the Fourteenth National Conference on Artificial Intelligence, AAAI Press/MIT Press, 1997, pp. 203–208.

[4] R. Bellman, Dynamic Programming, Princeton University Press, Princeton, NJ, 1957.

[5] A.L. Blum, M.L. Furst, Fast planning through planning graph analysis, Artificial Intelligence 90 (1–2) (1997) 279–298.

[6] A.L. Blum, J.C. Langford, Probabilistic planning in the Graphplan framework, in: Working Notes of the Workshop on Planning as Combinatorial Search, held in conjunction with the Fourth International Conference on Artificial Intelligence Planning, 1998.

[7] A.L. Blum, J.C. Langford, Probabilistic planning in the Graphplan framework, in: Proceedings of the Fifth European Conference on Planning, 1999, pp. 320–332.

[8] B. Bonet, H. Geffner, GPT: A tool for planning with uncertainty and partial information, in: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence Workshop on Planning under Uncertainty and Incomplete Information, 2001, http://citeseer.nj.nec.com/449314.html.
[9] C. Boutilier, T. Dean, A. Hanks, Decision-theoretic planning: Structural assumptions and computational leverage, J. Artificial Intelligence Res. 11 (1999) 1–94.
[10] C. Boutilier, R. Dearden, Approximating value trees in structured dynamic programming, in: L. Saitta (Ed.), Proceedings of the Thirteenth International Conference on Machine Learning, 1996, pp. 54–62.
[11] C. Boutilier, R. Dearden, M. Goldszmidt, Exploiting structure in policy construction, in: Proc. IJCAI-95, Montreal, Quebec, 1995, pp. 1104–1113.
[12] C. Boutilier, D. Poole, Computing optimal policies for partially observable decision processes using compact representations, in: Proceedings of The Thirteenth National Conference on Artificial Intelligence, AAAI Press/MIT Press, 1996, pp. 1168–1175.
[13] T. Bylander, The computational complexity of propositional STRIPS planning, Artificial Intelligence 69 (1994) 161–204.
[14] A. Cassandra, M.L. Littman, N.L. Zhang, Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes, in: Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-97), Morgan Kaufmann, San Francisco, CA, 1997, pp. 54–61, http://www.cs.duke.edu/mlittman/docs/uai97-pomdp.ps.
[15] A. Condon, The complexity of stochastic games, Inform. and Comput. 96 (2) (1992) 203–224.
[16] M. Davis, G. Logemann, D. Loveland, A machine program for theorem proving, Comm. ACM 5 (1962) 394–397.
[17] M. Davis, H. Putnam, A computing procedure for quantification theory, J. ACM 7 (1960) 201–215.
[18] E.V. Denardo, Dynamic Programming: Models and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1982.
[19] D. Draper, S. Hanks, D. Weld, Probabilistic planning with information gathering and contingent execution, in: Proceedings of the AAAI Spring Symposium on Decision Theoretic Planning, 1994, pp. 76–82.
[20] M.D. Ernst, T.D. Millstein, D.S. Weld, Automatic SAT-compilation of planning problems, in: Proc. IJCAI-97, Nagoya, Japan, 1997, pp. 1169–1176.
[21] R.E. Fikes, N.J. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, Artificial Intelligence 2 (1971) 189–208. Reprinted, in: J. Allen, J. Hendler, A. Tate (Eds.), Readings in Planning, Morgan Kaufmann, San Mateo, CA, 1990.
[22] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, San Francisco, CA, 1979.
[23] E. Giunchiglia, G. Neelakantan Kartha, V. Lifschitz, Representing action: Indeterminacy and ramifications, Artificial Intelligence 95 (2) (1997) 409–438, http://citeseer.nj.nec.com/giunchiglia97representing.html.
[24] R.P. Goldman, M.S. Boddy, Conditional linear planning, in: K. Hammond (Ed.), The Second International Conference on Artificial Intelligence Planning Systems, AAAI Press/MIT Press, 1994, pp. 80–85.
[25] R.P. Goldman, M.S. Boddy, Representing uncertainty in simple planners, in: Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR-94), 1994, pp. 238–245.
[26] S. Hanks, D. McDermott. Modeling, a dynamic and uncertain world I: Symbolic and probabilistic reasoning about change, Technical Report, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1993.
[27] E.A. Hansen, Z. Feng, Dynamic programming for POMDPs using a factored state representation, in: Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS), 2000, pp. 130–139, http://citeseer.nj.nec.com/hansen00dynamic.html.
[28] J. Hoey, R. St-Aubin, A. Hu, C. Boutilier, SPUDD: Stochastic planning using decision diagrams, in: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, 1999, pp. 279–288, http://citeseer.nj.nec.com/hoey99spudd.html.
[29] R.A. Howard, Dynamic Programming and Markov Processes, MIT Press, Cambridge, MA, 1960.
[30] L.P. Kaelbling, M.L. Littman, A.R. Cassandra, Planning and acting in partially observable stochastic domains, Artificial Intelligence 101 (1–2) (1998) 99–134.
[31] H. Kautz, D. McAllester, B. Selman, Encoding plans in propositional logic, in: Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR-96), 1996, pp. 374–385.

[32] H. Kautz, D. McAllester, B. Selman, Exploiting variable dependency in local search, in: Abstracts of the Poster Sessions of IJCAI-97, Nagoya, Japan, 1997.
[33] H. Kautz, B. Selman, Planning as satisfiability, in: Proceedings of Tenth European Conference on Artificial Inttelligence (ECAI-92), 1992, pp. 359–363.
[34] H. Kautz, B. Selman, Pushing the envelope: Planning, propositional logic, and stochastic search, in: Proceedings of the Thirteenth National Conference on Artificial Intelligence, AAAI Press/MIT Press, 1996, pp. 1194–1201.
[35] H. Kautz, B. Selman, BLACKBOX: A new approach to the application of theorem proving to problem solving, in: Working Notes of the Workshop on Planning as Combinatorial Search, 1998, pp. 58–60. Held in conjunction with the Fourth International Conference on Artificial Intelligence Planning.
[36] H. Kautz, B. Selman, The role of domain-specific knowledge in the planning as satisfiability framework, in: Proceedings of the Fourth International Conference on Artificial Intelligence Planning, AAAI Press, 1998, pp. 181–189.
[37] H. Kautz, B. Selman, Unifying SAT-based and graph-based planning, in: Proc. IJCAI-99, Stockholm, Sweden, 1999, pp. 318–325.
[38] M. Kearns, Y. Mansour, A.Y. Ng, A sparse sampling algorithm for near-optimal planning in large Markov decision processes, in: Proc. IJCAI-99, Stockholm, Sweden, AAAI Press/MIT Press, 1999, pp. 1324–1331.
[39] D. Koller, R. Parr, Computing factored value functions for policies in structured MDPs, in: Proc. IJCAI-99, Stockholm, Sweden, AAAI Press/MIT Press, 1999, pp. 1332–1339.
[40] D. Koller, R. Parr, Policy iteration for factored MDPs, in: Proceedings of the Sixteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI 2000), 2000.
[41] N. Kushmerick, S. Hanks, D.S. Weld, An algorithm for probabilistic planning, Artificial Intelligence 76 (1–2) (1995) 239–286.
[42] C.M. Li, Anbulagan, Heuristics based on unit propagation for satisfiability problems, in: Proc. IJCAI-97, Nagoya, Japan, 1997, pp. 366–371.
[43] M.L. Littman, Probabilistic propositional planning: Representations and complexity, in: Proceedings of the Fourteenth National Conference on Artificial Intelligence, AAAI Press/MIT Press, 1997, pp. 748–754, http://www.cs.duke.edu/mlittman/docs/aaai97-planning.ps.
[44] M.L. Littman, J. Goldsmith, M. Mundhenk, The computational complexity of probabilistic plan existence and evaluation, J. Artificial Intelligence Res. 9 (1998) 1–36.
[45] M.L. Littman, S.M. Majercik, T. Pitassi, Stochastic Boolean satisfiability, J. Automat. Reason. 27 (3) (2001) 251–296.
[46] O. Madani, S. Hanks, A. Condon, On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems, in: Proceedings of the Sixteenth National Conference on Artificial Intelligence, AAAI Press/MIT Press, 1999, pp. 541–548.
[47] S.M. Majercik, C-MAXPLAN: Contingent planning in the MAXPLAN framework, in: Proceedings of the AAAI Spring Symposium on Search Techniques for Problem Solving Under Uncertainty and Incomplete Information, Stanford, CA, 1999.
[48] S.M. Majercik, APROPOS^2: Approximate probabilistic planning out of stochastic satisfiability, in: Proceedings of the Eighteenth National Conference on Artificial Intelligence Workshop on Probabilistic Approaches in Search, 2002, To appear.
[49] S.M. Majercik, M.L. Littman, MAXPLAN: A new approach to probabilistic planning, in: R. Simmons, M. Veloso, S. Smith (Eds.), Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems, AAAI Press, 1998, pp. 86–93.
[50] S.M. Majercik, M.L. Littman, Using caching to solve larger probabilistic planning problems, in: Proceedings of the Fifteenth National Conference on Artificial Intelligence, AAAI Press/MIT Press, 1998, pp. 954–959.
[51] S.M. Majercik, A.P. Rusczek, Faster probabilistic planning through more efficient stochastic satisfiability problem encodings, in: Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems, AAAI Press, 2002.
[52] S.M. Majercik, Planning under uncertainty via stochastic satisfiability, PhD Thesis, Department of Computer Science, Duke University, September 2000.
[53] A.D. Mali, S. Kambhampati, On the utility of plan-space (causal) encodings, in: Proceedings of the Sixteenth National Conference on Artificial Intelligence, AAAI Press/MIT Press, 1999, pp. 557–563.

[54] D. McAllester, B. Selman, H. Kautz, Evidence for invariants in local search, in: Proceedings of the Fourteenth National Conference on Artificial Intelligence, AAAI Press/MIT Press, 1997, pp. 321–326.
[55] N.J. Nilsson, Principles of Artificial Intelligence, Tioga, Palo Alto, CA, 1980.
[56] N. Onder, Personal communication, 1998.
[57] N. Onder, Personal communication, 2000.
[58] N. Onder, M.E. Pollack, Contingency selection in plan generation, in: Proceedings of the Fourth European Conference on Planning: Recent Advances in AI Planning, 1997, pp. 364–376.
[59] N. Onder, M.E. Pollack, Conditional, probabilistic planning: A unifying algorithm and effective search control mechanisms, in: Proceedings of the Sixteenth National Conference on Artificial Intelligence, AAAI Press/MIT Press, 1999, pp. 577–584.
[60] C.H. Papadimitriou, Games against nature, J. Computer Systems Sci. 31 (1985) 288–301.
[61] C.H. Papadimitriou, Computational Complexity, Addison-Wesley, Reading, MA, 1994.
[62] M.A. Peot, D.E. Smith, Conditional nonlinear planning, in: Proceedings of the First International Conference on Artificial Intelligence Planning Systems, 1992, pp. 189–197.
[63] M.A. Peot, Decision-theoretic planning, PhD Thesis, Department of Engineering-Economic Systems and Operations Research, Stanford University, May 1998.
[64] L. Pryor, G. Collins, Planning for contingencies: A decision-based approach, J. Artificial Intelligence Res. 4 (1996) 287–339.
[65] M.L. Puterman, M.C. Shin, Modified policy iteration algorithms for discounted Markov decision processes, Management Sci. 24 (1978) 1127–1137.
[66] U. Schöning, A probabilistic algorithm for k-SAT and constraint satisfaction problems, in: Proceedings of the Fortieth Annual IEEE Symposium on Foundations of Computer Science, 1999, pp. 410–414.
[67] L. Schubert, Monotonic solution of the frame problem in the situation calculus; An efficient method for worlds with fully specified actions, in: H. Kyburg, R. Loui, G. Carlson (Eds.), Knowledge Representation and Defeasible Reasoning, Kluwer Academic, Dordrecht, 1990, pp. 23–67.
[68] B. Selman, H. Kautz, B. Cohen, Local search strategies for satisfiability testing, in: D.S. Johnson, M.A. Trick (Eds.), Cliques, Coloring, and Satisfiability, in: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26, American Mathematical Society, Providence, RI, 1996.
[69] D.E. Smith, D.S. Weld, Conformant Graphplan, in: Proceedings of the Fifteenth National Conference on Artificial Intelligence, AAAI Press/MIT Press, 1998, pp. 889–896.
[70] T. Walsh, Stochastic constraint programming, in: Proceedings of Fifteenth European Conference on Artificial Intelligence (ECAI-2002), 2002, To appear.
[71] D. Warren, Generating conditional plans and programs, in: Proceedings of the Summer Conference on AI and Simulation of Behavior, University of Edinburgh, 1976, pp. 344–354.
[72] D.S. Weld, C.R. Anderson, D.E. Smith, Extending Graphplan to handle uncertainty and sensing actions, in: Proceedings of the Fifteenth National Conference on Artificial Intelligence, AAAI Press/MIT Press, 1998, pp. 897–904.
[73] H. Zhang, M.E. Stickel, Implementing the Davis–Putnam method, J. Automat. Reason. 24 (1–3) (2000) 277–296.
[74] U. Zwick, M. Paterson, The complexity of mean payoff games on graphs, Theoret. Comput. Sci. 158 (1–2) (1996) 343–359.